Working Draft, Standard for Programming Language C++

Note: this is an early draft. It’s known to be incompleat and incorrekt, and it has lots of bad formatting.
## Contents

1 Scope 1

2 Normative references 2

3 Terms and definitions 3

4 General principles 10
  4.1 Implementation compliance 10
  4.2 Structure of this document 11
  4.3 Syntax notation 12

5 Lexical conventions 13
  5.1 Separate translation 13
  5.2 Phases of translation 13
  5.3 Character sets 14
  5.4 Preprocessing tokens 15
  5.5 Alternative tokens 16
  5.6 Tokens 16
  5.7 Comments 16
  5.8 Header names 17
  5.9 Preprocessing numbers 17
  5.10 Identifiers 17
  5.11 Keywords 18
  5.12 Operators and punctuators 19
  5.13 Literals 19

6 Basics 29
  6.1 Preamble 29
  6.2 Declarations and definitions 29
  6.3 One-definition rule 31
  6.4 Scope 35
  6.5 Name lookup 41
  6.6 Program and linkage 54
  6.7 Memory and objects 58
  6.8 Types 72
  6.9 Program execution 79

7 Expressions 91
  7.1 Preamble 91
  7.2 Properties of expressions 92
  7.3 Standard conversions 95
  7.4 Usual arithmetic conversions 99
  7.5 Primary expressions 100
  7.6 Compound expressions 116
  7.7 Constant expressions 148

8 Statements 154
  8.1 Preamble 154
  8.2 Labeled statement 155
  8.3 Expression statement 155
  8.4 Compound statement or block 155
  8.5 Selection statements 155
  8.6 Iteration statements 157
  8.7 Jump statements 160
8.8 Declaration statement .................................................. 162
8.9 Ambiguity resolution .................................................... 162

9 Declarations ........................................................................ 164
9.1 Preamble ........................................................................... 164
9.2 Specifiers .......................................................................... 166
9.3 Declarators ......................................................................... 183
9.4 Initializers .......................................................................... 198
9.5 Function definitions ........................................................... 214
9.6 Structured binding declarations ............................................ 220
9.7 Enumerations ...................................................................... 221
9.8 Namespaces ........................................................................ 224
9.9 The using declaration ......................................................... 231
9.10 The asm declaration ......................................................... 237
9.11 Linkage specifications .......................................................... 237
9.12 Attributes ......................................................................... 240

10 Modules ............................................................................. 248
10.1 Module units and purviews ..................................................... 248
10.2 Export declaration .............................................................. 249
10.3 Import declaration .............................................................. 252
10.4 Global module fragment ....................................................... 253
10.5 Private module fragment ....................................................... 255
10.6 Instantiation context ............................................................ 256
10.7 Reachability ....................................................................... 257

11 Classes ................................................................................. 259
11.1 Preamble ........................................................................... 259
11.2 Properties of classes ............................................................ 260
11.3 Class names ........................................................................ 261
11.4 Class members .................................................................... 263
11.5 Unions ................................................................................ 285
11.6 Local class declarations ........................................................ 288
11.7 Derived classes .................................................................... 288
11.8 Member name lookup ........................................................... 296
11.9 Member access control ........................................................ 299
11.10 Initialization ...................................................................... 308
11.11 Comparisons ..................................................................... 320
11.12 Free store ......................................................................... 323

12 Overloading ......................................................................... 325
12.1 Preamble ........................................................................... 325
12.2 Overloadable declarations ..................................................... 325
12.3 Declaration matching ............................................................ 327
12.4 Overload resolution ............................................................. 328
12.5 Address of overloaded function .............................................. 352
12.6 Overloaded operators .......................................................... 353
12.7 Built-in operators .................................................................. 357
12.8 User-defined literals ............................................................ 359

13 Templates ............................................................................ 361
13.1 Preamble ........................................................................... 361
13.2 Template parameters ........................................................... 362
13.3 Names of template specializations ......................................... 366
13.4 Template arguments ............................................................ 369
13.5 Template constraints .......................................................... 374
13.6 Type equivalence .................................................................. 380
13.7 Template declarations .......................................................... 381
13.8 Name resolution .................................................................. 402
13.9 Template instantiation and specialization ........................................ 418
13.10 Function template specializations .................................................. 431

14 Exception handling 452
14.1 Preamble .................................................................................... 452
14.2 Throwing an exception ................................................................... 453
14.3 Constructors and destructors ........................................................... 454
14.4 Handling an exception .................................................................... 455
14.5 Exception specifications ................................................................. 457
14.6 Special functions .......................................................................... 459

15 Preprocessing directives 461
15.1 Preamble .................................................................................... 461
15.2 Conditional inclusion ..................................................................... 463
15.3 Source file inclusion ....................................................................... 465
15.4 Module directive ........................................................................... 466
15.5 Header unit importation .................................................................. 467
15.6 Macro replacement ......................................................................... 468
15.7 Line control .................................................................................. 473
15.8 Error directive ............................................................................... 474
15.9Pragma directive ............................................................................ 474
15.10 Null directive .............................................................................. 474
15.11 Predefined macro names ............................................................... 474
15.12Pragma operator ........................................................................... 476

16 Library introduction 478
16.1 General ....................................................................................... 478
16.2 The C standard library .................................................................... 479
16.3 Method of description .................................................................... 479
16.4 Library-wide requirements .............................................................. 485

17 Language support library 506
17.1 General ....................................................................................... 506
17.2 Common definitions ...................................................................... 506
17.3 Implementation properties .............................................................. 510
17.4 Integer types ................................................................................ 520
17.5 Startup and termination ................................................................ 521
17.6 Dynamic memory management ..................................................... 523
17.7 Type identification ....................................................................... 530
17.8 Source location ............................................................................ 531
17.9 Exception handling ....................................................................... 533
17.10 Initializer lists ............................................................................. 537
17.11 Comparisons .............................................................................. 538
17.12 Coroutines .................................................................................... 546
17.13 Other runtime support .................................................................. 550

18 Concepts library 553
18.1 General ....................................................................................... 553
18.2 Equality preservation ..................................................................... 553
18.3 Header <concepts> synopsis .......................................................... 554
18.4 Language-related concepts .............................................................. 556
18.5 Comparison concepts ................................................................... 561
18.6 Object concepts ............................................................................ 564
18.7 Callable concepts ........................................................................ 564

19 Diagnostics library 566
19.1 General ....................................................................................... 566
19.2 Exception classes .......................................................................... 566
19.3 Assertions ..................................................................................... 569

Contents iv
25 Algorithms library 1044
  25.1 General .......................... 1044
  25.2 Algorithms requirements .......... 1044
  25.3 Parallel algorithms .......... 1046
  25.4 Header <algorithm> synopsis ...... 1049
  25.5 Algorithm result types ... 1084
  25.6 Non-modifying sequence operations 1087
  25.7 Mutating sequence operations ... 1099
  25.8 Sorting and related operations 1115
  25.9 Header <numeric> synopsis ...... 1141
  25.10 Generalized numeric operations 1145
  25.11 Specialized <memory> algorithms 1154
  25.12 C library algorithms ........ 1160

26 Numerics library 1161
  26.1 General .......................... 1161
  26.2 Numeric type requirements .......... 1161
  26.3 The floating-point environment 1161
  26.4 Complex numbers .................. 1162
  26.5 Bit manipulation .................. 1170
  26.6 Random number generation .......... 1173
  26.7 Numeric arrays .................. 1210
  26.8 Mathematical functions for floating-point types 1229
  26.9 Numbers .................. 1244

27 Time library 1245
  27.1 General .......................... 1245
  27.2 Header <chrono> synopsis .......... 1245
  27.3 Cpp17Clock requirements ........ 1259
  27.4 Time-related traits ........ 1259
  27.5 Class template duration .......... 1261
  27.6 Class template time_point .......... 1268
  27.7 Clocks .......................... 1271
  27.8 The civil calendar ........ 1281
  27.9 Class template hh_mm_ss .......... 1311
  27.10 12/24 hours functions .......... 1313
  27.11 Time zones .................. 1313
  27.12 Formatting .................. 1326
  27.13 Parsing .................. 1330
  27.14 Header <ctime> synopsis .......... 1334

28 Localization library 1335
  28.1 General .......................... 1335
  28.2 Header <locale> synopsis .......... 1335
  28.3 Locales .......................... 1336
  28.4 Standard locale categories 1342
  28.5 C library locales ........ 1374

29 Input/output library 1375
  29.1 General .......................... 1375
  29.2 Iostreams requirements .......... 1375
  29.3 Forward declarations .......... 1376
  29.4 Standard iostream objects 1378
  29.5 Iostreams base classes .......... 1379
  29.6 Stream buffers ........ 1395
  29.7 Formatting and manipulators 1403
  29.8 String-based streams .......... 1427
  29.9 File-based streams .......... 1441
  29.10 Synchronized output streams 1453

Contents vi
1 Scope

This document specifies requirements for implementations of the C++ programming language. The first such requirement is that they implement the language, so this document also defines C++. Other requirements and relaxations of the first requirement appear at various places within this document.

C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899:2018 Programming languages — C (hereinafter referred to as the C standard). C++ provides many facilities beyond those provided by C, including additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.
2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) ISO/IEC 2382, Information technology — Vocabulary
(1.2) ISO 8601:2004, Data elements and interchange formats — Information interchange — Representation of dates and times
(1.3) ISO/IEC 9899:2018, Programming languages — C
(1.4) ISO/IEC 9945:2003, Information Technology — Portable Operating System Interface (POSIX)
(1.5) ISO/IEC 10646, Information technology — Universal Coded Character Set (UCS)
(1.6) ISO/IEC 10646:2003, Information technology — Universal Multiple-Octet Coded Character Set (UCS)
(1.7) ISO 80000-2:2009, Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology

2 The library described in ISO/IEC 9899:2018, Clause 7, is hereinafter called the C standard library.
3 The operating system interface described in ISO/IEC 9945:2003 is hereinafter called POSIX.
4 The ECMAScript Language Specification described in Standard Ecma-262 is hereinafter called ECMA-262.
5 [Note 1: References to ISO/IEC 10646:2003 are used only to support deprecated features (D.21). — end note]
3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, the terms, definitions, and symbols given in ISO 80000-2:2009, and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

ISO Online browsing platform: available at https://www.iso.org/obp


Terms that are used only in a small portion of this document are defined where they are used and italicized where they are defined.

3.1 access
(execution-time action) read or modify the value of an object

[Note 1 to entry: Only objects of scalar type can be accessed. Reads of scalar objects are described in 7.3.2 and modifications of scalar objects are described in 7.6.19, 7.6.16, and 7.6.2.3. Attempts to read or modify an object of class type typically invoke a constructor (11.4.5) or assignment operator (11.4.6); such invocations do not themselves constitute accesses, although they may involve accesses of scalar subobjects. — end note]

3.2 arbitrary-positional stream
(library) stream that can seek to any integral position within the length of the stream

[Note 1 to entry: Every arbitrary-positional stream is also a reposition stream (3.48). — end note]

3.3 argument
(function call expression) expression in the comma-separated list bounded by the parentheses

3.4 argument
(function-like macro) sequence of preprocessing tokens in the comma-separated list bounded by the parentheses

3.5 argument
(throw expression) operand of throw

3.6 argument
(template instantiation) constant-expression, type-id, or id-expression in the comma-separated list bounded by the angle brackets

3.7 block
(execution) wait for some condition (other than for the implementation to execute the execution steps of the thread of execution) to be satisfied before continuing execution past the blocking operation

3.8 block
(statement) compound statement

3.9 character
(library) object which, when treated sequentially, can represent text

[Note 1 to entry: The term does not mean only char, char8_t, char16_t, char32_t, and wchar_t objects (6.8.2), but any value that can be represented by a type that provides the definitions specified in Clause 21, Clause 28, Clause 29, or Clause 30. — end note]
3.10 character container type
(library) class or a type used to represent a character

[Note 1 to entry: It is used for one of the template parameters of the string, iostream, and regular expression class templates. — end note]

3.11 collating element
sequence of one or more characters within the current locale that collate as if they were a single character

3.12 component
(library) group of library entities directly related as members, parameters, or return types

[Note 1 to entry: For example, the class template basic_string and the non-member function templates that operate on strings are referred to as the string component. — end note]

3.13 conditionally-supported
program construct that an implementation is not required to support

[Note 1 to entry: Each implementation documents all conditionally-supported constructs that it does not support. — end note]

3.14 constant subexpression
expression whose evaluation as subexpression of a conditional-expression CE would not prevent CE from being a core constant expression

3.15 deadlock
(library) situation wherein one or more threads are unable to continue execution because each is blocked waiting for one or more of the others to satisfy some condition

3.16 default behavior
(library implementation) specific behavior provided by the implementation, within the scope of the required behavior

3.17 diagnostic message
message belonging to an implementation-defined subset of the implementation’s output messages

3.18 direct-non-list-initialization
direct-initialization that is not list-initialization

3.19 dynamic type
(glvalue) type of the most derived object to which the glvalue refers

[Example 1: If a pointer (9.3.4.2) p whose static type is “pointer to class B” is pointing to an object of class D, derived from B (11.7), the dynamic type of the expression *p is “D”. References (9.3.4.3) are treated similarly. — end example]

3.20 dynamic type
(prvalue) static type of the prvalue expression
3.21 expression-equivalent
(library) expressions that all have the same effects, either are all potentially-throwing or are all not potentially-throwing, and either are all constant subexpressions or are all not constant subexpressions

[Example 1: For a value \( x \) of type \( \text{int} \) and a function \( f \) that accepts integer arguments, the expressions \( f(x + 2) \), \( f(2 + x) \), and \( f(1 + x + 1) \) are expression-equivalent. — end example]

3.22 finite state machine
(regular expression) unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained

3.23 format specifier
(regular expression) sequence of one or more characters that is to be replaced with some part of a regular expression match

3.24 handler function
(library) non-reserved function whose definition may be provided by a C++ program

[Note 1 to entry: A C++ program may designate a handler function at various points in its execution by supplying a pointer to the function when calling any of the library functions that install handler functions (Clause 17). — end note]

3.25 ill-formed program
program that is not well-formed (3.67)

3.26 implementation-defined behavior
behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents

3.27 implementation-defined strict total order over pointers
(library) implementation-defined strict total ordering over all pointer values such that the ordering is consistent with the partial order imposed by the builtin operators \(<\), \(>\), \(\leq\), \(\geq\), and \(\leq\geq\)

3.28 implementation limits
restrictions imposed upon programs by the implementation

3.29 iostream class templates
(library) templates that are declared in header <iosfwd> and take two template arguments

[Note 1 to entry: The arguments are named \( \text{charT} \) and \( \text{traits} \). The argument \( \text{charT} \) is a character container class, and the argument \( \text{traits} \) is a class which defines additional characteristics and functions of the character type represented by \( \text{charT} \) necessary to implement the iostream class templates. — end note]

3.30 locale-specific behavior
behavior that depends on local conventions of nationality, culture, and language that each implementation documents

3.31 matched
(regular expression) condition when a sequence of zero or more characters correspond to a sequence of characters defined by the pattern
3.32 modifier function
(library) class member function other than a constructor, assignment operator, or destructor that alters the state of an object of the class

3.33 move assignment
(library) assignment of an rvalue of some object type to a modifiable lvalue of the same type

3.34 move construction
(library) direct-initialization of an object of some type with an rvalue of the same type

3.35 multibyte character
sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment

[Note 1 to entry: The extended character set is a superset of the basic character set (5.3). — end note]

3.36 NTCTS
(library) sequence of values that have character type that precede the terminating null character type value charT()

3.37 observer function
(library) class member function that accesses the state of an object of the class but does not alter that state

[Note 1 to entry: Observer functions are specified as const member functions (11.4.3.2). — end note]

3.38 parameter
(function or catch clause) object or reference declared as part of a function declaration or definition or in the catch clause of an exception handler that acquires a value on entry to the function or handler

3.39 parameter
(function-like macro) identifier from the comma-separated list bounded by the parentheses immediately following the macro name

3.40 parameter
(template) member of a template-parameter-list

3.41 primary equivalence class
(regular expression) set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accents, case, or locale specific tailorings

3.42 program-defined specialization
(library) explicit template specialization or partial specialization that is not part of the C++ standard library and not defined by the implementation

3.43 program-defined type
(library) non-closure class type or enumeration type that is not part of the C++ standard library and not defined by the implementation, or a closure type of a non-implementation-provided lambda expression, or an instantiation of a program-defined specialization

[Note 1 to entry: Types defined by the implementation include extensions (4.1) and internal types used by the library. — end note]
3.44 projection

(library) transformation that an algorithm applies before inspecting the values of elements

[Example 1:

```cpp
std::pair<int, std::string_view> pairs[] = {{2, "foo"}, {1, "bar"}, {0, "baz"}};
std::ranges::sort(pairs, std::ranges::less{}, [] (auto const & p) { return p.first; });
```

sorts the pairs in increasing order of their first members:

```
{{0, "baz"}, {1, "bar"}, {2, "foo"}}
```

—end example]

3.45 referenceable type

(type) that is either an object type, a function type that does not have cv-qualifiers or a ref-qualifier, or a reference type

[Note 1 to entry: The term describes a type to which a reference can be created, including reference types. —end note]

3.46 regular expression

(pattern) that selects specific strings from a set of character strings

3.47 replacement function

(library) non-reserved function whose definition is provided by a C++ program

[Note 1 to entry: Only one definition for such a function is in effect for the duration of the program’s execution, as the result of creating the program (5.2) and resolving the definitions of all translation units (6.6). —end note]

3.48 repositional stream

(library) stream that can seek to a position that was previously encountered

3.49 required behavior

(library) description of replacement function and handler function semantics applicable to both the behavior provided by the implementation and the behavior of any such function definition in the program

[Note 1 to entry: If such a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined. —end note]

3.50 reserved function

(library) function, specified as part of the C++ standard library, that is defined by the implementation

[Note 1 to entry: If a C++ program provides a definition for any reserved function, the results are undefined. —end note]

3.51 signature

(function) name, parameter-type-list, and enclosing namespace (if any)

[Note 1 to entry: Signatures are used as a basis for name mangling and linking. —end note]

3.52 signature

(non-template friend function with trailing requires-clause) name, parameter-type-list, enclosing class, and trailing requires-clause

3.53 signature

(function template) name, parameter-type-list, enclosing namespace (if any), return type, template-head, and trailing requires-clause (if any)
3.54 signature
(Friend function template with constraint involving enclosing template parameters) name, parameter-type-list, return type, enclosing class, template-head, and trailing requires-clause (if any)

3.55 signature
(Function template specialization) signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced)

3.56 signature
(Class member function) name, parameter-type-list, class of which the function is a member, cv-qualifiers (if any), ref-qualifier (if any), and trailing requires-clause (if any)

3.57 signature
(Class member function template) name, parameter-type-list, class of which the function is a member, cv-qualifiers (if any), ref-qualifier (if any), return type (if any), template-head, and trailing requires-clause (if any)

3.58 signature
(Class member function template specialization) signature of the member function template of which it is a specialization and its template arguments (whether explicitly specified or deduced)

3.59 stable algorithm
(Library) algorithm that preserves, as appropriate to the particular algorithm, the order of elements

[Note 1 to entry: Requirements for stable algorithms are given in 16.4.6.8. — end note]

3.60 static type
Type of an expression resulting from analysis of the program without considering execution semantics

[Note 1 to entry: The static type of an expression depends only on the form of the program in which the expression appears, and does not change while the program is executing. — end note]

3.61 sub-expression
(Regular expression) subset of a regular expression that has been marked by parentheses

3.62 traits class
(Library) class that encapsulates a set of types and functions necessary for class templates and function templates to manipulate objects of types for which they are instantiated

3.63 unblock
Satisfy a condition that one or more blocked threads of execution are waiting for

3.64 undefined behavior
Behavior for which this document imposes no requirements

[Note 1 to entry: Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. Evaluation of a constant expression (7.7) never exhibits behavior explicitly specified as undefined in Clause 4 through Clause 15. — end note]
3.65 unspecified behavior
behavior, for a well-formed program construct and correct data, that depends on the implementation
[Note 1 to entry: The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this document. — end note]

3.66 valid but unspecified state
<library> value of an object that is not specified except that the object’s invariants are met and operations on the object behave as specified for its type
[Example 1: If an object \texttt{x} of type \texttt{std::vector<int>} is in a valid but unspecified state, \texttt{x.empty()} can be called unconditionally, and \texttt{x.front()} can be called only if \texttt{x.empty()} returns \texttt{false}. — end example]

3.67 well-formed program
C++ program constructed according to the syntax rules, diagnosable semantic rules, and the one-definition rule
4 General principles

4.1 Implementation compliance

4.1.1 General

1 The set of diagnosable rules consists of all syntactic and semantic rules in this document except for those rules containing an explicit notation that “no diagnostic is required” or which are described as resulting in “undefined behavior”.

2 Although this document states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:

(2.1) If a program contains no violations of the rules in Clause 5 through Clause 32 and Annex D, a conforming implementation shall, within its resource limits as described in Annex B, accept and correctly execute that program.

(2.2) If a program contains a violation of any diagnosable rule or an occurrence of a construct described in this document as “conditionally-supported” when the implementation does not support that construct, a conforming implementation shall issue at least one diagnostic message.

(2.3) If a program contains a violation of a rule for which no diagnostic is required, this document places no requirement on implementations with respect to that program.

[Note 1: During template argument deduction and substitution, certain constructs that in other contexts require a diagnostic are treated differently; see 13.10.3. — end note]

3 For classes and class templates, the library Clauses specify partial definitions. Private members (11.9) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library Clauses.

4 For functions, function templates, objects, and values, the library Clauses specify declarations. Implementations shall supply definitions consistent with the descriptions in the library Clauses.

5 The names defined in the library have namespace scope (9.8). A C++ translation unit (5.2) obtains access to these names by including the appropriate standard library header or importing the appropriate standard library named header unit (16.4.3.2).

6 The templates, classes, functions, and objects in the library have external linkage (6.6). The implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (5.2).

7 Two kinds of implementations are defined: a hosted implementation and a freestanding implementation. For a hosted implementation, this document defines the set of available libraries. A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that includes certain language-support libraries (16.4.2.4).

8 A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any well-formed program. Implementations are required to diagnose programs that use such extensions that are ill-formed according to this document. Having done so, however, they can compile and execute such programs.

9 Each implementation shall include documentation that identifies all conditionally-supported constructs that it does not support and defines all locale-specific characteristics.

4.1.2 Abstract machine

1 The semantic descriptions in this document define a parameterized nondeterministic abstract machine. This document places no requirement on the structure of conforming implementations. In particular, they need

5) “Correct execution” can include undefined behavior, depending on the data being processed; see Clause 3 and 6.9.1.

6) This documentation also defines implementation-defined behavior; see 4.1.2.
not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.\footnote{This provision is sometimes called the “as-if” rule, because an implementation is free to disregard any requirement of this document as long as the result is as if the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.}

2 Certain aspects and operations of the abstract machine are described in this document as implementation-defined (for example, \texttt{sizeof(int)}). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects.\footnote{This documentation also includes conditionally-supported constructs and locale-specific behavior. See 4.1.} Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the “corresponding instance” below).

3 Certain other aspects and operations of the abstract machine are described in this document as unspecified (for example, order of evaluation of arguments in a function call \((7.6.1.3))\). Where possible, this document defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution for a given program and a given input.

4 Certain other operations are described in this document as undefined (for example, the effect of attempting to modify a const object).

\begin{footnotesize}
\begin{itemize}
\item \textit{Note 1:} This document imposes no requirements on the behavior of programs that contain undefined behavior.
\item \textit{— end note}\end{itemize}
\end{footnotesize}

5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

6 The least requirements on a conforming implementation are:

\begin{footnotesize}
\begin{enumerate}
\item \textit{(6.1)} Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine.
\item \textit{(6.2)} At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.
\item \textit{(6.3)} The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.
\end{enumerate}
\end{footnotesize}

These collectively are referred to as the \textit{observable behavior} of the program.

\begin{footnotesize}
\begin{itemize}
\item \textit{Note 2:} More stringent correspondences between abstract and actual semantics can be defined by each implementation.
\item \textit{— end note}\end{itemize}
\end{footnotesize}

4.2 Structure of this document \footnote{4.2} [intro.structure]

1 Clause 5 through Clause 15 describe the C++ programming language. That description includes detailed syntactic specifications in a form described in 4.3. For convenience, Annex A repeats all such syntactic specifications.

2 Clause 17 through Clause 32 and Annex D (the library clauses) describe the C++ standard library. That description includes detailed descriptions of the entities and macros that constitute the library, in a form described in Clause 16.

3 Annex B recommends lower bounds on the capacity of conforming implementations.

4 Annex C summarizes the evolution of C++ since its first published description, and explains in detail the differences between C++ and C. Certain features of C++ exist solely for compatibility purposes; Annex D describes those features.
4.3 Syntax notation

In the syntax notation used in this document, syntactic categories are indicated by italic type, and literal words and characters in constant width type. Alternatives are listed on separate lines except in a few cases where a long set of alternatives is marked by the phrase “one of”. If the text of an alternative is too long to fit on a line, the text is continued on subsequent lines indented from the first one. An optional terminal or non-terminal symbol is indicated by the subscript “opt”, so

\{ expression_{opt} \}

indicates an optional expression enclosed in braces.

Names for syntactic categories have generally been chosen according to the following rules:

1. X-name is a use of an identifier in a context that determines its meaning (e.g., class-name, typedef-name).
2. X-id is an identifier with no context-dependent meaning (e.g., qualified-id).
3. X-seq is one or more X’s without intervening delimiters (e.g., declaration-seq is a sequence of declarations).
4. X-list is one or more X’s separated by intervening commas (e.g., identifier-list is a sequence of identifiers separated by commas).
5 Lexical conventions

5.1 Separate translation

The text of the program is kept in units called source files in this document. A source file together with all the headers (16.4.2.3) and source files included (15.3) via the preprocessing directive #include, less any source lines skipped by any of the conditional inclusion (15.2) preprocessing directives, is called a translation unit.

[Note 1: A C++ program need not all be translated at the same time. — end note]

[Note 2: Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (6.6) by (for example) calls to functions whose identifiers have external or module linkage, manipulation of objects whose identifiers have external or module linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program (6.6). — end note]

5.2 Phases of translation

The precedence among the syntax rules of translation is specified by the following phases.9

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary. The set of physical source file characters accepted is implementation-defined. Any source file character not in the basic source character set (5.3) is replaced by the universal-character-name that designates that character. An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a universal-character-name (e.g., using the \uXXXX notation), are handled equivalently except where this replacement is reverted (5.4) in a raw string literal.

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. Except for splices reverted in a raw string literal, if a splice results in a character sequence that matches the syntax of a universal-character-name, the behavior is undefined. A source file that is not empty and that does not end in a new-line character, or that ends in a new-line character immediately preceded by a backslash character before any such splicing takes place, shall be processed as if an additional new-line character were appended to the file.

3. The source file is decomposed into preprocessing tokens (5.4) and sequences of whitespace characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.10 Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of whitespace characters other than new-line is retained or replaced by one space character is unspecified. The process of dividing a source file's characters into preprocessing tokens is context-dependent.

[Example 1: See the handling of < within a #include preprocessing directive. — end example]

4. Preprocessing directives are executed, macro invocations are expanded, and _Pragma unary operator expressions are executed. If a character sequence that matches the syntax of a universal-character-name is produced by token concatenation (15.6.4), the behavior is undefined. A #include preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

5. Each basic source character set member in a character-literal or a string-literal, as well as each escape sequence and universal-character-name in a character-literal or a non-raw string literal, is converted to the corresponding member of the execution character set (5.13.3, 5.13.5); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.11

9) Implementations behave as if these separate phases occur, although in practice different phases can be folded together.

10) A partial preprocessing token would arise from a source file ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a header-name that is missing the closing “>”. A partial comment would arise from a source file ending with an unclosed “/*” comment.

11) An implementation need not convert all non-corresponding source characters to the same execution character.
6. Adjacent string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token (5.6). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit.

[Note 1: The process of analyzing and translating the tokens can occasionally result in one token being replaced by a sequence of other tokens (13.3). — end note]

It is implementation-defined whether the sources for module units and header units on which the current translation unit has an interface dependency (10.1, 10.3) are required to be available.

[Note 2: Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. — end note]

8. Translated translation units and instantiation units are combined as follows:

[Note 3: Some or all of these can be supplied from a library. — end note]

Each translated translation unit is examined to produce a list of required instantiations.

[Note 4: This can include instantiations which have been explicitly requested (13.9.3). — end note]

The definitions of the required templates are located. It is implementation-defined whether the source of the translation units containing these definitions is required to be available.

[Note 5: An implementation can choose to encode sufficient information into the translated translation unit so as to ensure the source is not required here. — end note]

All the required instantiations are performed to produce instantiation units.

[Note 6: These are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions. — end note]

The program is ill-formed if any instantiation fails.

9. All external entity references are resolved. Library components are linked to satisfy external references to entities not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

5.3 Character sets

1 The basic source character set consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:

```
 a b c d e f g h i j k l m n o p q r s t u v w x y z
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 0 1 2 3 4 5 6 7 8 9
 _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & |  
 ! = , " '
```

2 The universal-character-name construct provides a way to name other characters.

- hex-quad:
  - `hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit`

- `universal-character-name`:
  - `\u hex-quad`
  - `\U hex-quad hex-quad`

A universal-character-name designates the character in ISO/IEC 10646 (if any) whose code point is the hexadecimal number represented by the sequence of hexadecimal-digits in the universal-character-name. The program is ill-formed if that number is not a code point or if it is a surrogate code point. Noncharacter code points and reserved code points are considered to designate separate characters distinct from any ISO/IEC 10646 character. If a universal-character-name outside the c-char-sequence, s-char-sequence, or r-char-sequence of a character-literal or string-literal (in either case, including within a user-defined-literal) corresponds to a control character or to a character in the basic source character set, the program is ill-formed.13

12) The glyphs for the members of the basic source character set are intended to identify characters from the subset of ISO/IEC 10646 which corresponds to the ASCII character set. However, the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, and therefore implementations must document how the basic source characters are represented in source files.

13) A sequence of characters resembling a universal-character-name in an r-char-sequence (5.13.5) does not form a universal-character-name.

§ 5.3
The basic execution character set and the basic execution wide-character set shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a null character (respectively, null wide character), whose value is 0. For each basic execution character set, the values of the members shall be non-negative and distinct from one another. In both the source and execution basic character sets, the value of each character after 0 in the above list of decimal digits shall be one greater than the value of the previous. The execution character set and the execution wide-character set are implementation-defined supersets of the basic execution character set and the basic execution wide-character set, respectively. The values of the members of the execution character sets and the sets of additional members are locale-specific.

5.4 Preprocessing tokens

preprocessing-token:
  header-name
  import-keyword
  module-keyword
  export-keyword
  identifier
  pp-number
  character-literal
  user-defined-character-literal
  string-literal
  user-defined-string-literal
  preprocessing-op-or-punc
  each non-whitespace character that cannot be one of the above

1 Each preprocessing token that is converted to a token (5.6) shall have the lexical form of a keyword, an identifier, a literal, or an operator or punctuator.

2 A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: header names, placeholder tokens produced by preprocessing import and module directives (import-keyword, module-keyword, and export-keyword), identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and punctuators, and single non-whitespace characters that do not lexically match the other preprocessing token categories. If a ’ or a ” character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by whitespace; this consists of comments (5.7), or whitespace characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in Clause 15, in certain circumstances during translation phase 4, whitespace (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

3 If the input stream has been parsed into preprocessing tokens up to a given character:

(3.1) — If the next character begins a sequence of characters that could be the prefix and initial double quote of a raw string literal, such as R", the next preprocessing token shall be a raw string literal. Between the initial and final double quote characters of the raw string, any transformations performed in phases 1 and 2 (universal-character-names and line splicing) are reverted; this reversion shall apply before any d-char, r-char, or delimiting parenthesis is identified. The raw string literal is defined as the shortest sequence of characters that matches the raw-string pattern

  encoding-prefixopt R raw-string

(3.2) — Otherwise, if the next three characters are <:: and the subsequent character is neither : nor >, the < is treated as a preprocessing token by itself and not as the first character of the alternative token <::.

(3.3) — Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail, except that a header-name (5.8) is only formed

(3.3.1) — after the include or import preprocessing token in an #include (15.3) or import (15.5) directive, or

(3.3.2) — within a has-include-expression.
Example 1:
#define R "x"
const char* s = R"y";  // ill-formed raw string, not "x" "y"
—end example]

The import-keyword is produced by processing an import directive (15.5), the module-keyword is produced by preprocessing a module directive (15.4), and the export-keyword is produced by preprocessing either of the previous two directives.

[Note 1: None has any observable spelling. — end note]

Example 2: The program fragment 0xe+foo is parsed as a preprocessing number token (one that is not a valid integer-literal or floating-point-literal token), even though a parse as three preprocessing tokens 0xe, +, and foo might produce a valid expression (for example, if foo were a macro defined as 1). Similarly, the program fragment 1E1 is parsed as a preprocessing number (one that is a valid floating-point-literal token), whether or not E is a macro name. — end example]

Example 3: The program fragment x+++++y is parsed as x ++ ++ ++ y, which, if x and y have integral types, violates a constraint on increment operators, even though the parse x ++ ++ + y might yield a correct expression. — end example]

5.5 Alternative tokens [lex.digraph]

Alternative token representations are provided for some operators and punctuators. In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling. The set of alternative tokens is defined in Table 1.

Table 1: Alternative tokens [tab:lex.digraph]

<table>
<thead>
<tr>
<th>Alternative</th>
<th>Primary</th>
<th>Alternative</th>
<th>Primary</th>
<th>Alternative</th>
<th>Primary</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;%, }</td>
<td>and</td>
<td>%&gt;</td>
<td>bitor</td>
<td>&lt;:</td>
<td>xor</td>
</tr>
<tr>
<td>%&gt; }</td>
<td></td>
<td></td>
<td></td>
<td>&lt;:</td>
<td>or</td>
</tr>
<tr>
<td>&lt;:</td>
<td></td>
<td></td>
<td></td>
<td>:=</td>
<td>xor_eq</td>
</tr>
<tr>
<td>:=</td>
<td>xor_eq</td>
<td>%:</td>
<td>compl</td>
<td>#:</td>
<td>not_eq</td>
</tr>
<tr>
<td>%: #:</td>
<td>bitand</td>
<td>#:</td>
<td>&amp;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.6 Tokens [lex.token]

token:
  identifier
  keyword
  literal
  operator-or-punctuator

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, “whitespace”), as described below, are ignored except as they serve to separate tokens.

[Note 1: Some whitespace is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. — end note]

5.7 Comments [lex.comment]

The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates immediately before the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only whitespace characters shall appear between it and the new-line that terminates the comment; no diagnostic is required.

14) These include “digraphs” and additional reserved words. The term “digraph” (token consisting of two characters) is not perfectly descriptive, since one of the alternative preprocessing-tokens is %:%: and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren’t lexical keywords are colloquially known as “digraphs”.

15) Thus the “stringized” values (15.6.3) of [ and <: will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

16) Literals include strings and character and numeric literals.
[Note 1: The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment. — end note]

5.8 Header names

header-name:
   < h-char-sequence >

h-char-sequence:
   h-char
   h-char-sequence h-char

h-char:
   any member of the source character set except new-line and >

q-char-sequence:
   q-char
   q-char-sequence q-char

q-char:
   any member of the source character set except new-line and "

1 [Note 1: Header name preprocessing tokens only appear within a #include preprocessing directive, a __has_include preprocessing expression, or after certain occurrences of an import token (see 5.4). — end note]

The sequences in both forms of header-names are mapped in an implementation-defined manner to headers or to external source file names as specified in 15.3.

2 The appearance of either of the characters ' or \ or of either of the character sequences /* or // in a q-char-sequence or an h-char-sequence is conditionally-supported with implementation-defined semantics, as is the appearance of the character " in an h-char-sequence. 17

5.9 Preprocessing numbers

pp-number:
   digit
   . digit
   pp-number digit
   pp-number identifier-nondigit
   pp-number ' digit
   pp-number ' nondigit
   pp-number e sign
   pp-number E sign
   pp-number p sign
   pp-number P sign
   pp-number .

1 Preprocessing number tokens lexically include all integer-literal tokens (5.13.2) and all floating-point-literal tokens (5.13.4).

2 A preprocessing number does not have a type or a value; it acquires both after a successful conversion to an integer-literal token or a floating-point-literal token.

5.10 Identifiers

identifier:
   identifier-nondigit
   identifier identifier-nondigit
   identifier digit

identifier-nondigit:
   nondigit
   universal-character-name

17) Thus, a sequence of characters that resembles an escape sequence might result in an error, be interpreted as the character corresponding to the escape sequence, or have a completely different meaning, depending on the implementation.
An identifier is an arbitrarily long sequence of letters and digits. Each universal-character-name in an identifier shall designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in Table 2. The initial element shall not be a universal-character-name designating a character whose encoding falls into one of the ranges specified in Table 3. Upper- and lower-case letters are different. All characters are significant.18

Table 2: Ranges of characters allowed

<table>
<thead>
<tr>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00A8-00AF</td>
<td>Range 1</td>
</tr>
<tr>
<td>00B2-00B5</td>
<td>Range 2</td>
</tr>
<tr>
<td>00B7-00BA</td>
<td>Range 3</td>
</tr>
<tr>
<td>00C0-00D6</td>
<td>Range 4</td>
</tr>
<tr>
<td>00D8-00F6</td>
<td>Range 5</td>
</tr>
<tr>
<td>00F8-00FF</td>
<td>Range 6</td>
</tr>
<tr>
<td>0100-167F</td>
<td>Range 7</td>
</tr>
<tr>
<td>1681-180D</td>
<td>Range 8</td>
</tr>
<tr>
<td>180F-1FFF</td>
<td>Range 9</td>
</tr>
<tr>
<td>200B-200D</td>
<td>Range 10</td>
</tr>
<tr>
<td>202A-2040</td>
<td>Range 11</td>
</tr>
<tr>
<td>2070-218F</td>
<td>Range 12</td>
</tr>
<tr>
<td>2460-24FF</td>
<td>Range 13</td>
</tr>
<tr>
<td>2776-2793</td>
<td>Range 14</td>
</tr>
<tr>
<td>2C00-2DFF</td>
<td>Range 15</td>
</tr>
<tr>
<td>2E80-2FFF</td>
<td>Range 16</td>
</tr>
<tr>
<td>3004-3007</td>
<td>Range 17</td>
</tr>
<tr>
<td>3021-303F</td>
<td>Range 18</td>
</tr>
<tr>
<td>3031-D7FF</td>
<td>Range 19</td>
</tr>
<tr>
<td>F900-FD3D</td>
<td>Range 20</td>
</tr>
<tr>
<td>FD40-FDCF</td>
<td>Range 21</td>
</tr>
<tr>
<td>FDF0-FE44</td>
<td>Range 22</td>
</tr>
<tr>
<td>FE47-FFFD</td>
<td>Range 23</td>
</tr>
<tr>
<td>10000-1FFFFD</td>
<td>Range 24</td>
</tr>
<tr>
<td>20000-2FFFFD</td>
<td>Range 25</td>
</tr>
<tr>
<td>30000-3FFFFD</td>
<td>Range 26</td>
</tr>
<tr>
<td>40000-4FFFFD</td>
<td>Range 27</td>
</tr>
<tr>
<td>50000-5FFFFD</td>
<td>Range 28</td>
</tr>
<tr>
<td>60000-6FFFFD</td>
<td>Range 29</td>
</tr>
<tr>
<td>70000-7FFFFD</td>
<td>Range 30</td>
</tr>
<tr>
<td>80000-8FFFFD</td>
<td>Range 31</td>
</tr>
<tr>
<td>90000-9FFFFD</td>
<td>Range 32</td>
</tr>
<tr>
<td>A0000-AFFFD</td>
<td>Range 33</td>
</tr>
<tr>
<td>B0000-BFFFD</td>
<td>Range 34</td>
</tr>
<tr>
<td>C0000-CFFFFD</td>
<td>Range 35</td>
</tr>
<tr>
<td>D0000-DFFFFD</td>
<td>Range 36</td>
</tr>
<tr>
<td>E0000-EFFFFD</td>
<td>Range 37</td>
</tr>
</tbody>
</table>

Table 3: Ranges of characters disallowed initially (combining characters)

<table>
<thead>
<tr>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0300-036F</td>
<td>Range 1</td>
</tr>
<tr>
<td>1DC0-1DFF</td>
<td>Range 2</td>
</tr>
<tr>
<td>20D0-20FF</td>
<td>Range 3</td>
</tr>
<tr>
<td>FE20-FE2F</td>
<td>Range 4</td>
</tr>
</tbody>
</table>

2 The identifiers in Table 4 have a special meaning when appearing in a certain context. When referred to in the grammar, these identifiers are used explicitly rather than using the identifier grammar production. Unless otherwise specified, any ambiguity as to whether a given identifier has a special meaning is resolved to interpret the token as a regular identifier.

Table 4: Identifiers with special meaning

<table>
<thead>
<tr>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>final</td>
</tr>
<tr>
<td>import</td>
</tr>
<tr>
<td>module</td>
</tr>
<tr>
<td>override</td>
</tr>
</tbody>
</table>

3 In addition, some identifiers are reserved for use by C++ implementations and shall not be used otherwise; no diagnostic is required.

(3.1) Each identifier that contains a double underscore __ or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use.

(3.2) Each identifier that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

5.11 Keywords

keyword: any identifier listed in Table 5

import-keyword
module-keyword
export-keyword

18) On systems in which linkers cannot accept extended characters, an encoding of the universal-character-name can be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters can be used to encode the \u in a universal-character-name. Extended characters can produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers. In C++, upper- and lower-case letters are considered different for all identifiers, including external identifiers.
The identifiers shown in Table 5 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7) except in an attribute-token (9.12.1).

[Note 1: The register keyword is unused but is reserved for future use. — end note]

Table 5: Keywords [tabl:lex.key]

<table>
<thead>
<tr>
<th>alignas</th>
<th>constinit</th>
<th>false</th>
<th>public</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignof</td>
<td>const_cast</td>
<td>float</td>
<td>register</td>
<td>try</td>
</tr>
<tr>
<td>asm</td>
<td>continue</td>
<td>for</td>
<td>reinterpret_cast</td>
<td>typedef</td>
</tr>
<tr>
<td>auto</td>
<td>co_await</td>
<td>friend</td>
<td>requires</td>
<td>typeid</td>
</tr>
<tr>
<td>bool</td>
<td>co_return</td>
<td>goto</td>
<td>return</td>
<td>typename</td>
</tr>
<tr>
<td>break</td>
<td>co_yield</td>
<td>if</td>
<td>short</td>
<td>union</td>
</tr>
<tr>
<td>case</td>
<td>decltype</td>
<td>inline</td>
<td>signed</td>
<td>unsigned</td>
</tr>
<tr>
<td>catch</td>
<td>default</td>
<td>int</td>
<td>sizeof</td>
<td>using</td>
</tr>
<tr>
<td>char</td>
<td>delete</td>
<td>long</td>
<td>static</td>
<td>virtual</td>
</tr>
<tr>
<td>char8_t</td>
<td>do</td>
<td>mutable</td>
<td>static_assert</td>
<td>void</td>
</tr>
<tr>
<td>char16_t</td>
<td>double</td>
<td>namespace</td>
<td>static_cast</td>
<td>volatile</td>
</tr>
<tr>
<td>char32_t</td>
<td>dynamic_cast</td>
<td>new</td>
<td>struct</td>
<td>wchar_t</td>
</tr>
<tr>
<td>class</td>
<td>else</td>
<td>noexcept</td>
<td>switch</td>
<td>while</td>
</tr>
<tr>
<td>concept</td>
<td>enum</td>
<td>nullptr</td>
<td>template</td>
<td></td>
</tr>
<tr>
<td>const</td>
<td>explicit</td>
<td>operator</td>
<td>this</td>
<td></td>
</tr>
<tr>
<td>consteval</td>
<td>export</td>
<td>private</td>
<td>thread_local</td>
<td></td>
</tr>
<tr>
<td>constexpr</td>
<td>extern</td>
<td>protected</td>
<td>throw</td>
<td></td>
</tr>
</tbody>
</table>

Furthermore, the alternative representations shown in Table 6 for certain operators and punctuators (5.5) are reserved and shall not be used otherwise.

Table 6: Alternative representations [tabl:lex.key.digraph]

<table>
<thead>
<tr>
<th>and</th>
<th>and_eq</th>
<th>bitand</th>
<th>bitor</th>
<th>compl</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>not_eq</td>
<td>or</td>
<td>or_eq</td>
<td>xor</td>
<td>xor_eq</td>
<td></td>
</tr>
</tbody>
</table>

5.12 Operators and punctuators [lex.operators]

The lexical representation of C++ programs includes a number of preprocessing tokens that are used in the syntax of the preprocessor or are converted into tokens for operators and punctuators:

preprocessing-op-or-punc:
  preprocessing-operator
operator-or-punctuator

preprocessing-operator: one of
  
#       #
%

operator-or-punctuator: one of
  
{   }   [   ]   (   )
<:   <:   <%   %>   ;   :   ...
?    ::    .    .*   ->   -*   ~
!    +     -     *     /     %=  %=  ^=  &=  |=
==   ==   -=   ==   /=   %=  %=  &=  |=
==   !=   <     >   <=   >=   <<=  >>=  &k  |
<<   >>   <<=  >>=   ++   --   ,
and  or    xor    not    bitand  bitor  compl
and_eq  or_eq  xor_eq  not_eq

Each operator-or-punctuator is converted to a single token in translation phase 7 (5.2).

5.13 Literals [lex.literal]

5.13.1 Kinds of literals [lex.literal.kinds]

There are several kinds of literals.19

19) The term “literal” generally designates, in this document, those tokens that are called “constants” in ISO C.
literal:
  integer-literal
  character-literal
  floating-point-literal
  string-literal
  boolean-literal
  pointer-literal
  user-defined-literal

5.13.2 Integer literals

integer-literal:
  binary-literal integer-suffix opt
  octal-literal integer-suffix opt
  decimal-literal integer-suffix opt
  hexadecimal-literal integer-suffix opt

binary-literal:
  0b binary-digit
  0B binary-digit
  binary-literal 'opt binary-digit

octal-literal:
  0
  octal-literal 'opt octal-digit

decimal-literal:
  nonzero-digit
  decimal-literal 'opt digit

hexadecimal-literal:
  hexadecimal-prefix hexadecimal-digit-sequence

binary-digit: one of
  0 1

octal-digit: one of
  0 1 2 3 4 5 6 7
nonzero-digit: one of
  1 2 3 4 5 6 7 8 9

hexadecimal-prefix: one of
  0x 0X

hexadecimal-digit-sequence:
  hexadecimal-digit
  hexadecimal-digit-sequence 'opt hexadecimal-digit

hexadecimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-suffix:
  unsigned-suffix long-suffix opt
  unsigned-suffix long-long-suffix opt
  long-suffix unsigned-suffix opt
  long-long-suffix unsigned-suffix opt

unsigned-suffix: one of
  u U

long-suffix: one of
  l L

long-long-suffix: one of
  ll LL

1 In an integer-literal, the sequence of binary-digits, octal-digits, digits, or hexadecimal-digits is interpreted as a base $N$ integer as shown in table Table 7; the lexically first digit of the sequence of digits is the most significant.

[Note 1: The prefix and any optional separating single quotes are ignored when determining the value. — end note]
Table 7: Base of integer-literals

<table>
<thead>
<tr>
<th>Kind of integer-literal</th>
<th>base N</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary-literal</td>
<td>2</td>
</tr>
<tr>
<td>octal-literal</td>
<td>8</td>
</tr>
<tr>
<td>decimal-literal</td>
<td>10</td>
</tr>
<tr>
<td>hexadecimal-literal</td>
<td>16</td>
</tr>
</tbody>
</table>

2 The hexadecimal-digits a through f and A through F have decimal values ten through fifteen.

[Example 1: The number twelve can be written 12, 014, 0XC, or 0b1100. The integer-literals 1048576, 1’048’576, 0x100000, 0x10’0000, and 0’004’000’000 all have the same value. — end example]

3 The type of an integer-literal is the first type in the list in Table 8 corresponding to its optional integer-suffix in which its value can be represented. An integer-literal is a prvalue.

Table 8: Types of integer-literals

<table>
<thead>
<tr>
<th>integer-suffix</th>
<th>decimal-literal</th>
<th>integer-literal other than decimal-literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td>u or U</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>l or L</td>
<td>long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td>Both u or U and l or L</td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>ll or LL</td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td>Both u or U and ll or LL</td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>

4 If an integer-literal cannot be represented by any type in its list and an extended integer type (6.8.2) can represent its value, it may have that extended integer type. If all of the types in the list for the integer-literal are signed, the extended integer type shall be signed. If all of the types in the list for the integer-literal are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. A program is ill-formed if one of its translation units contains an integer-literal that cannot be represented by any of the allowed types.

5.13.3 Character literals

character-literal:

- encoding-prefixopt c-char-sequence

encoding-prefix: one of

u8 u U L

c-char-sequence:

- c-char
- c-char-sequence c-char

c-char:

- any member of the basic source character set except the single-quote ’, backslash \, or new-line character escape-sequence
- universal-character-name

§ 5.13.3
escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence: one of
  \'  \"  "  ?  \n  \a  \b  \f  \r  \t  \v

octal-escape-sequence:
  \ octal-digit
  \ octal-digit octal-digit
  \ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
  \x hexadecimal-digit
  hexadecimal-escape-sequence hexadecimal-digit

1 A character-literal that does not begin with u8, u, U, or L is an ordinary character literal. An ordinary character literal that contains a single c-char representable in the execution character set has type char, with value equal to the numerical value of the encoding of the c-char in the execution character set. An ordinary character literal that contains more than one c-char is a multicharacter literal. A multicharacter literal, or an ordinary character literal containing a single c-char not representable in the execution character set, is conditionally-supported, has type int, and has an implementation-defined value.

2 A character-literal that begins with u8, such as u8‘w’, is a character-literal of type char8_t, known as a UTF-8 character literal. The value of a UTF-8 character literal is equal to its ISO/IEC 10646 code point value, provided that the code point value can be encoded as a single UTF-8 code unit.

   [Note 1: That is, provided the code point value is in the range [0, 7F] (hexadecimal). — end note]

   If the value is not representable with a single UTF-8 code unit, the program is ill-formed. A UTF-8 character literal containing multiple c-chars is ill-formed.

3 A character-literal that begins with the letter u, such as u’x’, is a character-literal of type char16_t, known as a UTF-16 character literal. The value of a UTF-16 character literal is equal to its ISO/IEC 10646 code point value, provided that the code point value is representable with a single 16-bit code unit.

   [Note 2: That is, provided the code point value is in the range [0, FFFF] (hexadecimal). — end note]

   If the value is not representable with a single 16-bit code unit, the program is ill-formed. A UTF-16 character literal containing multiple c-chars is ill-formed.

4 A character-literal that begins with the letter U, such as U’y’, is a character-literal of type char32_t, known as a UTF-32 character literal. The value of a UTF-32 character literal containing a single c-char is equal to its ISO/IEC 10646 code point value. A UTF-32 character literal containing multiple c-chars is ill-formed.

5 A character-literal that begins with the letter L, such as L’z’, is a wide-character literal. A wide-character literal has type wchar_t. The value of a wide-character literal containing a single c-char has value equal to the numerical value of the encoding of the c-char in the execution wide-character set, unless the c-char has no representation in the execution wide-character set, in which case the value is implementation-defined.

   [Note 3: The type wchar_t is able to represent all members of the execution wide-character set (see 6.8.2). — end note]

   The value of a wide-character literal containing multiple c-chars is implementation-defined.

6 Certain non-graphic characters, the single quote ’, the double quote “, the question mark ?, and the backslash \, can be represented according to Table 9. The double quote “ and the question mark ?, can be represented as themselves or by the escape sequences \" and \? respectively, but the single quote ’ and the backslash \ shall be represented by the escape sequences \’ and \\ respectively. Escape sequences in which the character following the backslash is not listed in Table 9 are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

7 The escape \ooo consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \xhhh consists of the backslash followed by x followed by one or more hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is

---

20 They are intended for character sets where a character does not fit into a single byte.
21 Using an escape sequence for a question mark is supported for compatibility with ISO C++ 2014 and ISO C.

§ 5.13.3 22
Table 9: Escape sequences  [tab:lex.ccon.esc]

<table>
<thead>
<tr>
<th>Escape</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-line NL(LF)</td>
<td>\n</td>
</tr>
<tr>
<td>horizontal tab HT</td>
<td>\t</td>
</tr>
<tr>
<td>vertical tab VT</td>
<td>\v</td>
</tr>
<tr>
<td>backspace BS</td>
<td>\b</td>
</tr>
<tr>
<td>carriage return CR</td>
<td>\r</td>
</tr>
<tr>
<td>form feed FF</td>
<td>\f</td>
</tr>
<tr>
<td>alert BEL</td>
<td>\a</td>
</tr>
<tr>
<td>backslash \</td>
<td>\ \</td>
</tr>
<tr>
<td>question mark ?</td>
<td>?</td>
</tr>
<tr>
<td>single quote ’</td>
<td>\’</td>
</tr>
<tr>
<td>double quote &quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>octal number \ooo</td>
<td>\ooo</td>
</tr>
<tr>
<td>hex number \hhh</td>
<td>\xhhh</td>
</tr>
</tbody>
</table>

terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character-literal is implementation-defined if it falls outside of the implementation-defined range defined for char (for character-literals with no prefix) or wchar_t (for character-literals prefixed by L).

[Note 4: If the value of a character-literal prefixed by u, u8, or U is outside the range defined for its type, the program is ill-formed. — end note]

A universal-character-name is translated to the encoding, in the appropriate execution character set, of the character named. If there is no such encoding, the universal-character-name is translated to an implementation-defined encoding.

[Note 5: In translation phase 1, a universal-character-name is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in terms of universal-character-names. However, the actual compiler implementation can use its own native character set, so long as the same results are obtained. — end note]

5.13.4 Floating-point literals  [lex.fcon]

floating-point-literal:
  decimal-floating-point-literal
  hexadecimal-floating-point-literal
decimal-floating-point-literal:
  fractional-constant exponent-part opt floating-point-suffix opt
digit-sequence exponent-part floating-point-suffix opt
hexadecimal-floating-point-literal:
  hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-point-suffix opt
  hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-point-suffix opt
fractional-constant:
  digit-sequence opt . digit-sequence
digit-sequence .
hexadecimal-fractional-constant:
  hexadecimal-digit-sequence opt . hexadecimal-digit-sequence
  hexadecimal-digit-sequence .
exponent-part:
  e sign opt digit-sequence
  E sign opt digit-sequence
binary-exponent-part:
  p sign opt digit-sequence
  P sign opt digit-sequence
sign: one of
  + -
digit-sequence:
  digit
digit-sequence ‘ opt digit
The type of a floating-point-literal is determined by its floating-point-suffix as specified in Table 10.

Table 10: Types of floating-point-literals

<table>
<thead>
<tr>
<th>floating-point-suffix</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>double</td>
</tr>
<tr>
<td>f or F</td>
<td>float</td>
</tr>
<tr>
<td>l or L</td>
<td>long double</td>
</tr>
</tbody>
</table>

The significand of a floating-point-literal is the fractional-constant or digit-sequence of a decimal-floating-point-literal or the hexadecimal-fractional-constant or hexadecimal-digit-sequence of a hexadecimal-floating-point-literal. In the significand, the sequence of digits or hexadecimal-digits and optional period are interpreted as a base $N$ real number $s$, where $N$ is 10 for a decimal-floating-point-literal and 16 for a hexadecimal-floating-point-literal.

[Note 1: Any optional separating single quotes are ignored when determining the value. — end note]

If an exponent-part or binary-exponent-part is present, the exponent $e$ of the floating-point-literal is the result of interpreting the sequence of an optional sign and the digits as a base 10 integer. Otherwise, the exponent $e$ is 0. The scaled value of the literal is $s \times 10^e$ for a decimal-floating-point-literal and $s \times 2^e$ for a hexadecimal-floating-point-literal.

[Example 1: The floating-point-literals 49.625 and 0xC.68p+2 have the same value. The floating-point-literals 1.602’176’565e-19 and 1.602176565e-19 have the same value. — end example]

If the scaled value is not in the range of representable values for its type, the program is ill-formed. Otherwise, the value of a floating-point-literal is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner.

### 5.13.5 String literals

- **string-literal:**
  - encoding-prefix-opt " s-char-sequence-opt "
  - encoding-prefix-opt R raw-string
- **s-char-sequence:**
  - s-char
  - s-char-sequence s-char
- **s-char:**
  - any member of the basic source character set except the double-quote "," backslash \\, or new-line character
  - escape-sequence
  - universal-character-name
- **raw-string:**
  - " d-char-sequence-opt ( r-char-sequence-opt ) d-char-sequence-opt "
- **r-char-sequence:**
  - r-char
  - r-char-sequence r-char
- **r-char:**
  - any member of the source character set, except a right parenthesis ) followed by the initial d-char-sequence (which may be empty) followed by a double quote ".
- **d-char-sequence:**
  - d-char
  - d-char-sequence d-char
- **d-char:**
  - any member of the basic source character set except:
    - space, the left parenthesis (, the right parenthesis ), the backslash \\, and the control characters representing horizontal tab, vertical tab, form feed, and newline.

A string-literal that has an R in the prefix is a raw string literal. The d-char-sequence serves as a delimiter. The terminating d-char-sequence of a raw-string is the same sequence of characters as the initial d-char-sequence. A d-char-sequence shall consist of at most 16 characters.
[Note 1: The characters ‘{’ and ‘}’ are permitted in a raw-string. Thus, `R"delimiter((a|b))delimiter"` is equivalent to "(a|b)". — end note]

[Note 2: A source-file new-line in a raw string literal results in a new-line in the resulting execution string literal. Assuming no whitespace at the beginning of lines in the following example, the assert will succeed:

```cpp
const char* p = R"(a\n)\nab\nab\nab\n)ab";
assert(std::strcmp(p, "a\nab\nab\nab\n") == 0);
— end note]

[Example 1: The raw string

\[ R"a(\n\nab)\nab\n)ab" \]

is equivalent to "\n\na\nab\nab\n\n). The raw string

\[ R"x = \"y\"\"\" \]

is equivalent to "x = \n\ny\n\n\n. — end example]

After translation phase 6, a string-literal that does not begin with an encoding-prefix is an ordinary string literal. An ordinary string literal has type “array of n const char” where n is the size of the string as defined below, has static storage duration (6.7.5), and is initialized with the given characters.

A string-literal that begins with u8, such as `u8"asdf"`, is a UTF-8 string literal. A UTF-8 string literal has type “array of n const char8_t”, where n is the size of the string as defined below; each successive element of the object representation (6.8) has the value of the corresponding code unit of the UTF-8 encoding of the string.

Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals.

A string-literal that begins with u, such as `u"asdf"`, is a UTF-16 string literal. A UTF-16 string literal has type “array of n const char16_t”, where n is the size of the string as defined below; each successive element of the array has the value of the corresponding code unit of the UTF-16 encoding of the string.

[Note 3: A single c-char may produce more than one char16_t character in the form of surrogate pairs. A surrogate pair is a representation for a single code point as a sequence of two 16-bit code units. — end note]

A string-literal that begins with U, such as `U"asdf"`, is a UTF-32 string literal. A UTF-32 string literal has type “array of n const char32_t”, where n is the size of the string as defined below; each successive element of the array has the value of the corresponding code unit of the UTF-32 encoding of the string.

A string-literal that begins with L, such as `L"asdf"`, is a wide string literal. A wide string literal has type “array of n const wchar_t”, where n is the size of the string as defined below; it is initialized with the given characters.

In translation phase 6 (5.2), adjacent string-literals are concatenated. If both string-literals have the same encoding-prefix, the resulting concatenated string-literal has that encoding-prefix. If one string-literal has no encoding-prefix, it is treated as a string-literal of the same encoding-prefix as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally-supported with implementation-defined behavior.

[Note 4: This concatenation is an interpretation, not a conversion. Because the interpretation happens in translation phase 6 (after each character from a string-literal has been translated into a value from the appropriate character set), a string-literal’s initial rawness has no effect on the interpretation or well-formedness of the concatenation. — end note]

Table 11 has some examples of valid concatenations.

<table>
<thead>
<tr>
<th>Source</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>u&quot;a&quot;</td>
<td>u&quot;b&quot;</td>
</tr>
<tr>
<td>u&quot;a&quot;</td>
<td>&quot;b&quot;</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>&quot;b&quot;</td>
</tr>
<tr>
<td>Source</td>
<td>Means</td>
</tr>
<tr>
<td>u&quot;a&quot;</td>
<td>U&quot;b&quot;</td>
</tr>
<tr>
<td>U&quot;a&quot;</td>
<td>&quot;b&quot;</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>&quot;b&quot;</td>
</tr>
</tbody>
</table>

Table 11: String literal concatenations

§ 5.13.5
Characters in concatenated strings are kept distinct.

[Example 2: 
"\xa" "B"
contains the two characters 'A' and 'B' after concatenation (and not the single hexadecimal character '\xAB').
— end example]

12 After any necessary concatenation, in translation phase 7 (5.2), 'O' is appended to every string-literal so that programs that scan a string can find its end.

13 Escape sequences and universal-character-names in non-raw string literals have the same meaning as in character-literals (5.13.3), except that the single quote ' is representable either by itself or by the escape sequence \, and the double quote " shall be preceded by a \, and except that a universal-character-name in a UTF-16 string literal may yield a surrogate pair. In a narrow string literal, a universal-character-name may map to more than one char or char8_t element due to multibyte encoding. The size of a char16_t or wide string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for the terminating U'O' or L'O'. The size of a UTF-16 string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for each character requiring a surrogate pair, plus one for the terminating u'O'.

[Note 5: The size of a char16_t string literal is the number of code units, not the number of characters. — end note]

[Note 6: Any universal-character-names are required to correspond to a code point in the range [0, D800) or [E000, 10FFFF] (hexadecimal) (5.3). — end note]

The size of a narrow string literal is the total number of escape sequences and other characters, plus at least one for the multibyte encoding of each universal-character-name, plus one for the terminating 'O'.

14 Evaluating a string-literal results in a string literal object with static storage duration, initialized from the given characters as specified above. Whether all string-literals are distinct (that is, are stored in nonoverlapping objects) and whether successive evaluations of a string-literal yield the same or a different object is unspecified.

[Note 7: The effect of attempting to modify a string-literal is undefined. — end note]

5.13.6 Boolean literals

boolean-literal:
false
true

1 The Boolean literals are the keywords false and true. Such literals are prvalues and have type bool.

5.13.7 Pointer literals

pointer-literal:
nnullptr

1 The pointer literal is the keyword nullptr. It is a prvalue of type std::nullptr_t.

[Note 1: std::nullptr_t is a distinct type that is neither a pointer type nor a pointer-to-member type; rather, a prvalue of this type is a null pointer constant and can be converted to a null pointer value or null member pointer value. See 7.3.12 and 7.3.13. — end note]

5.13.8 User-defined literals

user-defined-literal:
user-defined-integer-literal
user-defined-floating-point-literal
user-defined-string-literal
user-defined-character-literal
user-defined-integer-literal:
decimal-literal ud-suffix
octal-literal ud-suffix
hexadecimal-literal ud-suffix
binary-literal ud-suffix
user-defined-floating-point-literal:
fractional-constant exponent-part_opt ud-suffix
digit-sequence exponent-part ud-suffix
hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part ud-suffix
hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part ud-suffix

§ 5.13.8
user-defined-string-literal:
  string-literal ud-suffix

user-defined-character-literal:
  character-literal ud-suffix

ud-suffix:
  identifier

1 If a token matches both user-defined-literal and another literal kind, it is treated as the latter.
Example 1: 123_km is a user-defined-literal, but 12LL is an integer-literal. — end example
The syntactic non-terminal preceding the ud-suffix in a user-defined-literal is taken to be the longest sequence
of characters that could match that non-terminal.

2 A user-defined-literal is treated as a call to a literal operator or literal operator template (12.8). To determine
the form of this call for a given user-defined-literal L with ud-suffix X, the literal-operator-id whose literal suffix
identifier is X is looked up in the context of L using the rules for unqualified name lookup (6.5.2). Let S be
the set of declarations found by this lookup. S shall not be empty.

3 If L is a user-defined-integer-literal, let n be the literal without its ud-suffix. If S contains a literal operator
with parameter type unsigned long long, the literal L is treated as a call of the form
  operator "" X(nULL)
Otherwise, S shall contain a raw literal operator or a numeric literal operator template (12.8) but not both.
If S contains a raw literal operator, the literal L is treated as a call of the form
  operator "" X("n")
Otherwise (S contains a numeric literal operator template), L is treated as a call of the form
  operator "" X<\texttt{c}_1, \texttt{c}_2, \ldots, \texttt{c}_k>()
where n is the source character sequence \texttt{c}_1\texttt{c}_2...\texttt{c}_k.
[Note 1: The sequence \texttt{c}_1\texttt{c}_2...\texttt{c}_k can only contain characters from the basic source character set. — end note]

4 If L is a user-defined-floating-point-literal, let f be the literal without its ud-suffix. If S contains a literal
operator with parameter type long double, the literal L is treated as a call of the form
  operator "" X(fL)
Otherwise, S shall contain a raw literal operator or a numeric literal operator template (12.8) but not both.
If S contains a raw literal operator, the literal L is treated as a call of the form
  operator "" X("f")
Otherwise (S contains a numeric literal operator template), L is treated as a call of the form
  operator "" X<\texttt{c}_1, \texttt{c}_2, \ldots, \texttt{c}_k>()
where f is the source character sequence \texttt{c}_1\texttt{c}_2...\texttt{c}_k.
[Note 2: The sequence \texttt{c}_1\texttt{c}_2...\texttt{c}_k can only contain characters from the basic source character set. — end note]

5 If L is a user-defined-string-literal, let str be the literal without its ud-suffix and let len be the number of
code units in str (i.e., its length excluding the terminating null character). If S contains a literal operator
template with a non-type template parameter for which str is a well-formed template-argument, the literal L
is treated as a call of the form
  operator "" X<str>()
Otherwise, the literal L is treated as a call of the form
  operator "" X(str, len)

6 If L is a user-defined-character-literal, let ch be the literal without its ud-suffix. S shall contain a literal
operator (12.8) whose only parameter has the type of ch and the literal L is treated as a call of the form
  operator "" X(ch)

Example 2:
  long double operator "" _w(long double);
  std::string operator "" _w(const char16_t*, std::size_t);
  unsigned operator "" _w(const char*);
  int main() {
      1.2_w; // calls operator "" _w(1.2L)

§ 5.13.8 27
In translation phase 6 (5.2), adjacent string-literals are concatenated and user-defined-string-literals are considered string-literals for that purpose. During concatenation, ud-suffixes are removed and ignored and the concatenation process occurs as described in 5.13.5. At the end of phase 6, if a string-literal is the result of a concatenation involving at least one user-defined-string-literal, all the participating user-defined-string-literals shall have the same ud-suffix and that suffix is applied to the result of the concatenation.

Example 3:

```c
int main() {
    L"A" "B" "C"_x;  // OK: same as L"ABC"_x
    "P"_x "Q" "R"_y;  // error: two different ud-suffixes
}
```

— end example]
6 Basics

6.1 Preamble

[Note 1: This Clause presents the basic concepts of the C++ language. It explains the difference between an object and a name and how they relate to the value categories for expressions. It introduces the concepts of a declaration and a definition and presents C++’s notion of type, scope, linkage, and storage duration. The mechanisms for starting and terminating a program are discussed. Finally, this Clause presents the fundamental types of the language and lists the ways of constructing compound types from these. — end note]

[Note 2: This Clause does not cover concepts that affect only a single part of the language. Such concepts are discussed in the relevant Clauses. — end note]

An entity is a value, object, reference, structured binding, function, enumerator, type, class member, bit-field, template, template specialization, namespace, or pack.

A name is a use of an identifier (5.10), operator-function-id (12.6), literal-operator-id (12.8), conversion-function-id (11.4.8.3), or template-id (13.3) that denotes an entity or label (8.7.6, 8.2).

Every name that denotes an entity is introduced by a declaration. Every name that denotes a label is introduced either by a goto statement (8.7.6) or a labeled-statement (8.2).

A variable is introduced by the declaration of a reference other than a non-static data member or of an object. The variable’s name, if any, denotes the reference or object.

A local entity is a variable with automatic storage duration (6.7.5.4), a structured binding (9.6) whose corresponding variable is such an entity, or the *this object (7.5.2).

Some names denote types or templates. In general, whenever a name is encountered it is necessary to determine whether that name denotes one of these entities before continuing to parse the program that contains it. The process that determines this is called name lookup (6.5).

Two names are the same if

— they are identifiers composed of the same character sequence, or
— they are operator-function-ids formed with the same operator, or
— they are conversion-function-ids formed with the same type, or
— they are template-ids that refer to the same class, function, or variable (13.6), or
— they are literal-operator-ids (12.8) formed with the same literal suffix identifier.

A name used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (6.6) of the name specified in each translation unit.

6.2 Declarations and definitions

A declaration (Clause 9) may introduce one or more names into a translation unit or redefine names introduced by previous declarations. If so, the declaration specifies the interpretation and semantic properties of these names. A declaration may also have effects including:

— a static assertion (9.1),
— controlling template instantiation (13.9.3),
— guiding template argument deduction for constructors (13.7.2.3),
— use of attributes (9.12), and
— nothing (in the case of an empty-declaration).

Each entity declared by a declaration is also defined by that declaration unless:

— it declares a function without specifying the function’s body (9.5),
— it contains the extern specifier (9.2.2) or a linkage-specification (9.11) and neither an initializer nor a function-body,

22) Appearing inside the brace-enclosed declaration-seq in a linkage-specification does not affect whether a declaration is a definition.
— it declares a non-inline static data member in a class definition (11.4, 11.4.9),
— it declares a static data member outside a class definition and the variable was defined within the class with the constexpr specifier (this usage is deprecated; see D.7),
— it is introduced by an elaborated-type-specifier (11.3),
— it is an opaque-enum-declaration (9.7.1),
— it is a template-parameter (13.2),
— it is a parameter-declaration (9.3.4.6) in a function declarator that is not the declarator of a function-definition,
— it is a typedef declaration (9.2.4),
— it is an alias-declaration (9.2.4),
— it is a using-declaration (9.9),
— it is a deduction-guide (13.7.2.3),
— it is a static_assert-declaration (9.1),
— it is an attribute-declaration (9.1),
— it is an empty-declaration (9.1),
— it is a using-directive (9.8.4),
— it is a using-enum-declaration (9.7.2),
— it is a template-declaration (13.1) whose template-head is not followed by either a concept-definition or a declaration that defines a function, a class, a variable, or a static data member.
— it is an explicit instantiation declaration (13.9.3), or
— it is an explicit specialization (13.9.4) whose declaration is not a definition.

A declaration is said to be a definition of each entity that it defines.

[Example 1: All but one of the following are definitions:

```
int a; // defines a
extern const int c = 1; // defines c
int f(int x) { return x+a; } // defines f and defines x
struct S { int a; int b; }; // defines S, S::a, and S::b
struct X {
    int x; // defines non-static data member x
    static int y; // declares static data member y
    X() : x(0) {} // defines a constructor of X
};
int X::y = 1; // defines X::y
enum { up, down }; // defines up and down
namespace N { int d; } // defines N and N::d
namespace N1 = N; // defines N1
X anX; // defines anX
```

whereas these are just declarations:

```
extern int a; // declares a
extern const int c; // declares c
int f(int); // declares f
struct S; // declares S
typedef int Int; // declares Int
extern X anotherX; // declares anotherX
using N::d; // declares d
```

— end example]

[Note 1: In some circumstances, C++ implementations implicitly define the default constructor (11.4.5.2), copy constructor, move constructor (11.4.5.3), copy assignment operator, move assignment operator (11.4.6), or destructor (11.4.7) member functions. — end note]

[Example 2: Given

```
#include <string>
```
struct C {
    std::string s;                  // std::string is the standard library class (21.3)
};

int main() {
    C a;
    C b = a;
    b = a;
}

the implementation will implicitly define functions to make the definition of C equivalent to

struct C {
    std::string s;
    C() : s() { }                     // s(std::move(x.s)) { }
    C(const C& x) : s(x.s) { }        // s = x.s; return *this;
    C(C&& x) : s(static_cast<std::string&&>(x.s)) { }
    C& operator=(const C& x) { s = x.s; return *this; }
    C& operator=(C&& x) { s = static_cast<std::string&&>(x.s); return *this; }
    ~C() { }                         // s = std::move(x.s); return *this;
};

—end example

[Note 2: A class name can also be implicitly declared by an elaborated-type-specifier (9.2.9.4). — end note]

In the definition of an object, the type of that object shall not be an incomplete type (6.8), an abstract class type (11.7.4), or a (possibly multi-dimensional) array thereof.

6.3 One-definition rule  [basic.def.odr]

1 No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, template, default argument for a parameter (for a function in a given scope), or default template argument.

2 An expression or conversion is potentially evaluated unless it is an unevaluated operand (7.2), a subexpression thereof, or a conversion in an initialization or conversion sequence in such a context. The set of potential results of an expression $E$ is defined as follows:

1. If $E$ is an id-expression (7.5.4), the set contains only $E$.
2. If $E$ is a subscripting operation (7.6.1.2) with an array operand, the set contains the potential results of that operand.
3. If $E$ is a class member access expression (7.6.1.5) of the form $E_1 . template_{opt} E_2$ naming a non-static data member, the set contains the potential results of $E_1$.
4. If $E$ is a class member access expression naming a static data member, the set contains the id-expression designating the data member.
5. If $E$ is a pointer-to-member expression (7.6.4) of the form $E_1 .* E_2$, the set contains the potential results of $E_1$.
6. If $E$ has the form $(E_1)$, the set contains the potential results of $E_1$.
7. If $E$ is a glvalue conditional expression (7.6.16), the set is the union of the sets of potential results of the second and third operands.
8. If $E$ is a comma expression (7.6.20), the set contains the potential results of the right operand.
9. Otherwise, the set is empty.

[Note 1: This set is a (possibly-empty) set of id-expressions, each of which is either $E$ or a subexpression of $E$.

[Example 1: In the following example, the set of potential results of the initializer of $n$ contains the first $S::x$ subexpression, but not the second $S::x$ subexpression.

```cpp
struct S { static const int x = 0; }
const int k = (const int &r);
int n = b ? (1, S::x) : f(S::x);       // S::x is odr-used here
```

§ 6.3
A function is named by an expression or conversion as follows:

(3.1) A function is named by an expression or conversion if it is the selected member of an overload set (6.5, 12.4, 12.5) in an overload resolution performed as part of forming that expression or conversion, unless it is a pure virtual function and either the expression is not an id-expression naming the function with an explicitly qualified name or the expression forms a pointer to member (7.6.2.2).

[Note 2: This covers taking the address of functions (7.3.4, 7.6.2.2), calls to named functions (7.6.1.3), operator overloading (Clause 12), user-defined conversions (11.4.8.3), allocation functions for new-expressions (7.6.2.8), as well as non-default initialization (9.4). A constructor selected to copy or move an object of class type is considered to be named by an expression or conversion even if the call is actually elided by the implementation (11.10.6). — end note]

(3.2) A deallocation function for a class is named by a new-expression if it is the single matching deallocation function for the allocation function selected by overload resolution, as specified in 7.6.2.8.

(3.3) A deallocation function for a class is named by a delete-expression if it is the selected usual deallocation function as specified in 7.6.2.9 and 11.12.

A variable \(x\) whose name appears as a potentially-evaluated expression \(E\) is odr-used by \(E\) unless

(4.1) \(x\) is a reference that is usable in constant expressions (7.7), or

(4.2) \(x\) is a variable of non-reference type that is usable in constant expressions and has no mutable subobjects, and \(E\) is an element of the set of potential results of an expression of non-volatile-qualified non-class type to which the lvalue-to-rvalue conversion (7.3.2) is applied, or

(4.3) \(x\) is a variable of non-reference type, and \(E\) is an element of the set of potential results of a discarded-value expression (7.2) to which the lvalue-to-rvalue conversion is not applied.

A structured binding is odr-used if it appears as a potentially-evaluated expression.

*\(\texttt{this}\) is odr-used if \(\texttt{this}\) appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (11.4.3)).

A virtual member function is odr-used if it is not pure. A function is odr-used if it is named by a potentially-evaluated expression or conversion. A non-placement allocation or deallocation function for a class is odr-used by the definition of a constructor of that class. A non-placement deallocation function for a class is odr-used by the definition of the destructor of that class, or by being selected by the lookup at the point of definition of a virtual destructor (11.4.7).²³

An assignment operator function in a class is odr-used by an implicitly-defined copy-assignment or move-assignment function for another class as specified in 11.4.6. A constructor for a class is odr-used as specified in 9.4. A destructor for a class is odr-used if it is potentially invoked (11.4.7).

A local entity (6.1) is odr-usable in a declarative region (6.4.1) if:

(9.1) either the local entity is not \(\texttt{\ast this}\), or an enclosing class or non-lambda function parameter scope exists and, if the innermost such scope is a function parameter scope, it corresponds to a non-static member function, and

(9.2) for each intervening declarative region (6.4.1) between the point at which the entity is introduced and the region (where \(\texttt{\ast this}\) is considered to be introduced within the innermost enclosing class or non-lambda function definition scope), either:

(9.2.1) the intervening declarative region is a block scope, or

(9.2.2) the intervening declarative region is the function parameter scope of a lambda-expression that has a simple-capture naming the entity or has a capture-default, and the block scope of the lambda-expression is also an intervening declarative region.

If a local entity is odr-used in a declarative region in which it is not odr-usable, the program is ill-formed.

[Example 2:

```c
void f(int n) {
    [] { n = 1; }; // error: n is not odr-usable due to intervening lambda-expression
```

²³ An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.
Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement (8.5.2); no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 11.4.5.2, 11.4.5.3, 11.4.7, and 11.4.6).

Example 3:

```c
auto f() {
    struct A {};
    return A{};
}
decltype(f()) g();
auto x = g();
```

A program containing this translation unit is ill-formed because `g` is odr-used but not defined, and cannot be defined in any other translation unit because the local class `A` cannot be named outside this translation unit. — end example]

A definition domain is a private-module-fragment or the portion of a translation unit excluding its private-module-fragment (if any). A definition of an inline function or variable shall be reachable from the end of every definition domain in which it is odr-used outside of a discarded statement.

A definition of a class shall be reachable in every context in which the class is used in a way that requires the class type to be complete.

Example 4: The following complete translation unit is well-formed, even though it never defines `X`:

```c
struct X;
struct X* x1;          // declare X as a struct type
X* x2;                 // use X in pointer formation
— end example]
```

Note 3: The rules for declarations and expressions describe in which contexts complete class types are required. A class type `T` must be complete if:

12.1 — an object of type `T` is defined (6.2), or
12.2 — a non-static class data member of type `T` is declared (11.4), or
12.3 — `T` is used as the allocated type or array element type in a new-expression (7.6.2.8), or
12.4 — an lvalue-to-rvalue conversion is applied to a glvalue referring to an object of type `T` (7.3.2), or
12.5 — an expression is converted (either implicitly or explicitly) to type `T` (7.3.2, 7.6.1.4, 7.6.1.7, 7.6.1.9, 7.6.3), or
12.6 — an expression that is not a null pointer constant, and has type other than `cv void*`, is converted to the type pointer to `T` or reference to `T` using a standard conversion (7.3), a dynamic_cast (7.6.1.7) or a static_cast (7.6.1.9), or
12.7 — a class member access operator is applied to an expression of type `T` (7.6.1.5), or
12.8 — the typeid operator (7.6.1.8) or the sizeof operator (7.6.2.5) is applied to an operand of type `T`, or
12.9 — a function with a return type or argument type of type `T` is defined (6.2) or called (7.6.1.3), or
12.10 — a class with a base class of type `T` is defined (11.7), or
12.11 — an lvalue of type `T` is assigned to (7.6.19), or
12.12 — the type `T` is the subject of an alignof expression (7.6.2.6), or
12.13 — an exception-declaration has type `T`, reference to `T`, or pointer to `T` (14.4).

— end note]

There can be more than one definition of a

13.1 — class type (Clause 11),
13.2 — enumeration type (9.7.1),
inline function or variable (9.2.8),
— templated entity (13.1),
— default argument for a parameter (for a function in a given scope) (9.3.4.7), or
— default template argument (13.2)
in a program provided that each definition appears in a different translation unit and the definitions satisfy the following requirements. Given such an entity \( D \) defined in more than one translation unit, for all definitions of \( D \), or, if \( D \) is an unnamed enumeration, for all definitions of \( D \) that are reachable at any given program point, the following requirements shall be satisfied.

— Each such definition shall not be attached to a named module (10.1).
— Each such definition shall consist of the same sequence of tokens, where the definition of a closure type is considered to consist of the sequence of tokens of the corresponding lambda-expression.
— In each such definition, corresponding names, looked up according to 6.5, shall refer to the same entity, after overload resolution (12.4) and after matching of partial template specialization (13.10.4), except that a name can refer to
  — a non-volatile const object with internal or no linkage if the object has the same literal type in all definitions of \( D \),
  — is initialized with a constant expression (7.7),
  — is not odr-used in any definition of \( D \), and
  — has the same value in all definitions of \( D \),
  or
  — a reference with internal or no linkage initialized with a constant expression such that the reference refers to the same entity in all definitions of \( D \).
— In each such definition, except within the default arguments and default template arguments of \( D \), corresponding lambda-expressions shall have the same closure type (see below).
— In each such definition, corresponding entities shall have the same language linkage.
— In each such definition, the overloaded operators referred to, the implicit calls to conversion functions, constructors, operator new functions and operator delete functions, shall refer to the same function.
— In each such definition, a default argument used by an (implicit or explicit) function call or a default template argument used by an (implicit or explicit) template-id or simple-template-id is treated as if its token sequence were present in the definition of \( D \); that is, the default argument or default template argument is subject to the requirements described in this paragraph (recursively).
— If \( D \) is a class with an implicitly-declared constructor (11.4.5.2, 11.4.5.3), it is as if the constructor was implicitly defined in every translation unit where it is odr-used, and the implicit definition in every translation unit shall call the same constructor for a subobject of \( D \).

[Example 5]:

```cpp
// translation unit 1:
struct X {
    X(int, int);
    X(int, int, int);
};
X::X(int, int = 0) { }
class D {
    X x = 0;
};
D d1; // X(int, int) called by D()

// translation unit 2:
struct X {
    X(int, int);
    X(int, int, int);
};
X::X(int, int = 0, int = 0) { }
```
class D {
    X x = 0;
};
D d2;                 // X(int, int, int) called by D();
                      // D()'s implicit definition violates the ODR

— end example]   (13.15) —
If D is a class with a defaulted three-way comparison operator function (11.11.3), it is as if the operator
was implicitly defined in every translation unit where it is odr-used, and the implicit definition in every
translation unit shall call the same comparison operators for each subobject of D.

If D is a template and is defined in more than one translation unit, then the preceding requirements shall
apply both to names from the template’s enclosing scope used in the template definition (13.8.4), and also
to dependent names at the point of instantiation (13.8.3). These requirements also apply to corresponding
entities defined within each definition of D (including the closure types of lambda-expressions, but excluding
entities defined within default arguments or default template arguments of either D or an entity not defined
within D). For each such entity and for D itself, the behavior is as if there is a single entity with a single
definition, including in the application of these requirements to other entities.

[Note 4: The entity is still declared in multiple translation units, and 6.6 still applies to these declarations. In
particular, lambda-expressions (7.5.5) appearing in the type of D might result in the different declarations having
distinct types, and lambda-expressions appearing in a default argument of D might still denote different types in
different translation units. — end note]

15 If these definitions do not satisfy these requirements, then the program is ill-formed; a diagnostic is required
only if the entity is attached to a named module and a prior definition is reachable at the point where a later
definition occurs.

16 [Example 6: inline void f(bool cond, void (*p)()) {
    if (cond) f(false, []());
} inline void g(bool cond, void (*p)() = []()) {
    if (cond) g(false);
} struct X {
    void h(bool cond, void (*p)() = []()) {
        if (cond) h(false);
    }
};
If the definition of f appears in multiple translation units, the behavior of the program is as if there is only one
definition of f. If the definition of g appears in multiple translation units, the program is ill-formed (no diagnostic
required) because each such definition uses a default argument that refers to a distinct lambda-expression closure type.
The definition of X can appear in multiple translation units of a valid program; the lambda-expressions defined within
the default argument of X::h within the definition of X denote the same closure type in each translation unit. — end
example]

If, at any point in the program, there is more than one reachable unnamed enumeration definition in the same
scope that have the same first enumerator name and do not have typedef names for linkage purposes (9.7.1),
those unnamed enumeration types shall be the same; no diagnostic required.

6.4 Scope [basic.scope]
6.4.1 Declarative regions and scopes [basic.scope.declarative]

Every name is introduced in some portion of program text called a declarative region, which is the largest part
of the program in which that name is valid, that is, in which that name may be used as an unqualified name
to refer to the same entity. In general, each particular name is valid only within some possibly discontiguous
portion of program text called its scope. To determine the scope of a declaration, it is sometimes convenient
to refer to the potential scope of a declaration. The scope of a declaration is the same as its potential scope
unless the potential scope contains another declaration of the same name. In that case, the potential scope of
the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in
the outer (containing) declarative region.

[Example 1: In
int j = 24;
int main() {
    int i = j, j;
    j = 42;
}

the identifier \texttt{j} is declared twice as a name (and used twice). The declarative region of the first \texttt{j} includes the entire example. The potential scope of the first \texttt{j} begins immediately after that \texttt{j} and extends to the end of the program, but its (actual) scope excludes the text between the \{ and the \}. The declarative region of the second declaration of \texttt{j} (the \texttt{j} immediately before the semicolon) includes all the text between \{ and \}, but its potential scope excludes the declaration of \texttt{i}. The scope of the second declaration of \texttt{j} is the same as its potential scope. — end example]

3 The names declared by a declaration are introduced into the scope in which the declaration occurs, except that the presence of a \texttt{friend} specifier (11.9.4), certain uses of the \texttt{elaborated-type-specifier} (9.2.9.4), and \texttt{using-directives} (9.8.4) alter this general behavior.

4 Given a set of declarations in a single declarative region, each of which specifies the same unqualified name,

(4.1) — they shall all refer to the same entity, or all refer to functions and function templates; or

(4.2) — exactly one declaration shall declare a class name or enumeration name that is not a typedef name
and the other declarations shall all refer to the same variable, non-static data member, or enumerator,
or all refer to functions and function templates; in this case the class name or enumeration name is
hidden (6.4.10).

[Note 1: A structured binding (9.6), namespace name (9.8), or class template name (13.1) must be unique in
its declarative region. — end note]

[Note 2: These restrictions apply to the declarative region into which a name is introduced, which is not necessarily
the same as the region in which the declaration occurs. In particular, \texttt{elaborated-type-specifiers} (9.2.9.4) and friend
declarations (11.9.4) can introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply
to that region. Local extern declarations (6.6) can introduce a name into the declarative region where the declaration
appears and also introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to both
regions. — end note]

5 For a given declarative region \texttt{R} and a point \texttt{P} outside \texttt{R}, the set of \texttt{intervening} declarative regions between
\texttt{P} and \texttt{R} comprises all declarative regions that are or enclose \texttt{R} and do not enclose \texttt{P}.

6 [Note 3: The name lookup rules are summarized in 6.5. — end note]

6.4.2 Point of declaration [basic.scope.pdecl]

1 The \textit{point of declaration} for a name is immediately after its complete declarator (9.3) and before its \texttt{initializer}
(if any), except as noted below.

[Example 1:

\begin{verbatim}
    unsigned char x = 12;
    { unsigned char x = x; }
\end{verbatim}

Here, the initialization of the second \texttt{x} has undefined behavior, because the initializer accesses the second \texttt{x} outside its
lifetime (6.7.3). — end example]

2 [Note 1: A name from an outer scope remains visible up to the point of declaration of the name that hides it.

[Example 2:

\begin{verbatim}
    const int i = 2;
    { int i[1]; }
\end{verbatim}

declares a block-scope array of two integers. — end example]

— end note]

3 The point of declaration for a class or class template first declared by a \texttt{class-specifier} is immediately
after the \texttt{identifier} or \texttt{simple-template-id} (if any) in its \texttt{class-head} (11.1). The point of declaration for an
enumeration is immediately after the \texttt{identifier} (if any) in either its \texttt{enum-specifier} (9.7.1) or its first \texttt{opaque-enum-declaration}
(9.7.1), whichever comes first. The point of declaration of an alias or alias template
immediately follows the \texttt{defining-type-id} to which the alias refers.

4 The point of declaration of a \texttt{using-declarator} that does not name a constructor is immediately after the
\texttt{using-declarator} (9.9).

5 The point of declaration for an enumerator is immediately after its \texttt{enumerator-definition}.
Example 3:
```c
const int x = 12;
{ enum { x = x }; }
```
Here, the enumerator `x` is initialized with the value of the constant `x`, namely 12. — end example]

After the point of declaration of a class member, the member name can be looked up in the scope of its class. [Note 2: This is true even if the class is an incomplete class. For example,
```c
struct X {
    enum E { z = 16 };
    int b[X::z]; // OK
};
```
— end note]

7 The point of declaration of a class first declared in an elaborated-type-specifier is as follows:

(7.1) — for a declaration of the form
class-key attribute-specifier-seq\textsubscript{opt} identifier ;
the identifier is declared to be a class-name in the scope that contains the declaration, otherwise

(7.2) — for an elaborated-type-specifier of the form
class-key identifier if the elaborated-type-specifier is used in the decl-specifier-seq or parameter-declaration-clause of a function defined in namespace scope, the identifier is declared as a class-name in the namespace that contains the declaration; otherwise, except as a friend declaration, the identifier is declared in the smallest namespace or block scope that contains the declaration.

[Note 3: These rules also apply within templates. — end note]

[Note 4: Other forms of elaborated-type-specifier do not declare a new name, and therefore must refer to an existing type-name. See 6.5.5 and 9.2.9.4. — end note]

8 The point of declaration for an injected-class-name (11.1) is immediately following the opening brace of the class definition.

9 The point of declaration for a function-local predefined variable (9.5.1) is immediately before the function-body of a function definition.

10 The point of declaration of a structured binding (9.6) is immediately after the identifier-list of the structured binding declaration.

11 The point of declaration for the variable or the structured bindings declared in the for-range-declaration of a range-based for statement (8.6.5) is immediately after the for-range-initializer.

12 The point of declaration for a template parameter is immediately after its complete template-parameter.

Example 4:
```c
typedef unsigned char T;
template<class T = T // lookup finds the typedef name of unsigned char
    struct A { T N = 0> struct A { }; // lookup finds the template parameter
``` — end example]

[Note 5: Friend declarations refer to functions or classes that are members of the nearest enclosing namespace, but they do not introduce new names into that namespace (9.8.2.3). Function declarations at block scope and variable declarations with the extern specifier at block scope refer to declarations that are members of an enclosing namespace, but they do not introduce new names into that scope. — end note]

[Note 6: For point of instantiation of a template, see 13.8.5.1. — end note]

6.4.3 Block scope

A name declared in a block (8.4) is local to that block; it has block scope. Its potential scope begins at its point of declaration (6.4.2) and ends at the end of its block. A variable declared at block scope is a local variable.
The name declared in an exception-declaration is local to the handler and shall not be redeclared in the outermost block of the handler.

Names declared in the init-statement, the for-range-declaration, and in the condition of if, while, for, and switch statements are local to the if, while, for, or switch statement (including the controlled statement), and shall not be redeclared in a subsequent condition of that statement nor in the outermost block (or, for the if statement, any of the outermost blocks) of the controlled statement.

[Example 1:

```c
if (int x = f()) {
    int x;        // error: redeclaration of x
}
else {
    int x;        // error: redeclaration of x
}
@end example]

6.4.4 Function parameter scope

A function parameter (including one appearing in a lambda-declarator) or function-local predefined variable (9.5) has function parameter scope. The potential scope of a parameter or function-local predefined variable begins at its point of declaration. If the nearest enclosing function declarator is not the declarator of a function definition, the potential scope ends at the end of that function declarator. Otherwise, if the function has a function-try-block the potential scope ends at the end of the last associated handler. Otherwise the potential scope ends at the end of the outermost block of the function definition. A parameter name shall not be redeclared in the outermost block of the function definition nor in the outermost block of any handler associated with a function-try-block.

6.4.5 Function scope

Labels (8.2) have function scope and may be used anywhere in the function in which they are declared. Only labels have function scope.

6.4.6 Namespace scope

The declarative region of a namespace-definition is its namespace-body. Entities declared in a namespace-body are said to be members of the namespace, and names introduced by these declarations into the declarative region of the namespace are said to be member names of the namespace. A namespace member name has namespace scope. Its potential scope includes its namespace from the name’s point of declaration (6.4.2) onwards; and for each using-directive (9.8.4) that nominates the member’s namespace, the member’s potential scope includes that portion of the potential scope of the using-directive that follows the member’s point of declaration.

[Example 1:

```c
namespace N {
    int i;
    int g(int a) { return a; }
    int j();
    void q();
}

namespace { int l=1; }
// the potential scope of l is from its point of declaration to the end of the translation unit

namespace N {
    int g(char a) { // overloads N::g(int)
        return i+a;  // l is from unnamed namespace
    }

    int i;         // error: duplicate definition
    int j();      // OK: duplicate function declaration

    int j() {      // OK: definition of N::j()
        return g(i); // calls N::g(int)
    }
```
int q();  // error: different return type

—end example]

2 If a translation unit \( Q \) is imported into a translation unit \( R \) (10.3), the potential scope of a name \( X \) declared with namespace scope in \( Q \) is extended to include the portion of the corresponding namespace scope in \( R \) following the first \textit{module-import-declaration} or \textit{module-declaration} in \( R \) that imports \( Q \) (directly or indirectly) if

(2.1) — \( X \) does not have internal linkage, and

(2.2) — \( X \) is declared after the \textit{module-declaration} in \( Q \) (if any), and

(2.3) — either \( X \) is exported or \( Q \) and \( R \) are part of the same module.

\[ \text{Note 1: A module-import-declaration} \text{ imports both the named translation unit(s) and any modules named by exported module-import-declarations within them, recursively.} \]

\[ \text{Example 2:} \]

Translation unit #1:

\begin{verbatim}
export module Q;
export int sq(int i) { return i*i; }
\end{verbatim}

Translation unit #2:

\begin{verbatim}
export module R;
export import Q;
\end{verbatim}

Translation unit #3:

\begin{verbatim}
import R;
int main() { return sq(9); }  // OK: sq from module Q
\end{verbatim}

—end example]

—end note]

3 A namespace member can also be referred to after the :: scope resolution operator (7.5.4.3) applied to the name of its namespace or the name of a namespace which nominates the member’s namespace in a \textit{using-directive}; see 6.5.4.3.

4 The outermost declarative region of a translation unit is also a namespace, called the \textit{global namespace}. A name declared in the global namespace has \textit{global namespace scope} (also called \textit{global scope}). The potential scope of such a name begins at its point of declaration (6.4.2) and ends at the end of the translation unit that is its declarative region. A name with global namespace scope is said to be a \textit{global name}.

6.4.7 Class scope [basic.scope.class]

1 The potential scope of a name declared in a class consists not only of the declarative region following the name’s point of declaration, but also of all complete-class contexts (11.4) of that class.

2 A name \( N \) used in a class \( S \) shall refer to the same declaration in its context and when re-evaluated in the completed scope of \( S \). No diagnostic is required for a violation of this rule.

3 A name declared within a member function hides a declaration of the same name whose scope extends to or past the end of the member function’s class.

4 The potential scope of a declaration in a class that extends to or past the end of a class definition also extends to the regions defined by its member definitions, even if the members are defined lexically outside the class (this includes static data member definitions, nested class definitions, and member function definitions, including the member function body and any portion of the declarator part of such definitions which follows the \textit{declarator-id}, including a \textit{parameter-declaration-clause} and any default arguments (9.3.4.7)).

5 [Example 1:]

\begin{verbatim}
typedef int c;
enum { i = 1 };

class X {
  char v[i];  // error: i refers to ::i but when reevaluated is X::i
  int f() { return sizeof(c); }  // OK: X::c
  char c;
\end{verbatim}
```c
enum { i = 2 };  

typedef char* T;  
struct Y { 
    T a;  
    typedef long T;  
    T b;  
};  

typedef int I;  
class D { 
    typedef I I;  
};  

—end example
```

6 The name of a class member shall only be used as follows:

- in the scope of its class (as described above) or a class derived (11.7) from its class,
- after the . operator applied to an expression of the type of its class (7.6.1.5) or a class derived from its class,
- after the -> operator applied to a pointer to an object of its class (7.6.1.5) or a class derived from its class,
- after the :: scope resolution operator (7.5.4.3) applied to the name of its class or a class derived from its class.

6.4.8 Enumeration scope [basic.scope.enum]

1 The name of a scoped enumerator (9.7.1) has enumeration scope. Its potential scope begins at its point of declaration and terminates at the end of the enum-specifier.

6.4.9 Template parameter scope [basic.scope.temp]

1 The declarative region of the name of a template parameter of a template template-parameter is the smallest template-parameter-list in which the name was introduced.

2 The declarative region of the name of a template parameter of a template is the smallest template-declaration in which the name was introduced. Only template parameter names belong to this declarative region; any other kind of name introduced by the declaration of a template-declaration is instead introduced into the same declarative region where it would be introduced as a result of a non-template declaration of the same name.

[Example 1]:
```c
namespace N { 
    template<class T> struct A { };  
    template<class U> void f(U) { }  
    struct B { 
        template<class V> friend int g(struct C*);  
    };  
}  
```

The declarative regions of T, U and V are the template-declarations on lines #1, #2, and #3, respectively. But the names A, f, g and C all belong to the same declarative region — namely, the namespace-body of N. (g is still considered to belong to this declarative region in spite of its being hidden during qualified and unqualified name lookup.) — end example]

3 The potential scope of a template parameter name begins at its point of declaration (6.4.2) and ends at the end of its declarative region.

[Note 1: This implies that a template-parameter can be used in the declaration of subsequent template-parameters and their default arguments but cannot be used in preceding template-parameters or their default arguments. For example,]
```c
template<class T, T* p, class U = T> class X { /* ... */ };  
template<class T> void f(T* p = new T);  
```

This also implies that a template-parameter can be used in the specification of base classes. For example,
```c
template<class T> class X : public Array<T> { /* ... */ };  
```
template<class T> class Y : public T { /* ... */ };

The use of a template parameter as a base class implies that a class used as a template argument must be defined and not just declared when the class template is instantiated. — end note

The declarative region of the name of a template parameter is nested within the immediately-enclosing declarative region.

[Note 2: As a result, a template-parameter hides any entity with the same name in an enclosing scope (6.4.10).
Example 2:

typedef int N;
template<N X, typename N, template<N Y> class T> struct A;
Here, X is a non-type template parameter of type int and Y is a non-type template parameter of the same type as the second template parameter of A. — end example]

— end note]

[Note 3: Because the name of a template parameter cannot be redeclared within its potential scope (13.8.2), a template parameter’s scope is often its potential scope. However, it is still possible for a template parameter name to be hidden; see 13.8.2. — end note]

6.4.10 Name hiding [basic.scope.hiding]

1 A declaration of a name in a nested declarative region hides a declaration of the same name in an enclosing declarative region; see 6.4.1 and 6.5.2.

2 If a class name (11.3) or enumeration name (9.7.1) and a variable, data member, function, or enumerator are declared in the same declarative region (in any order) with the same name (excluding declarations made visible via using-directives (6.5.2)), the class or enumeration name is hidden wherever the variable, data member, function, or enumerator name is visible.

3 In a member function definition, the declaration of a name at block scope hides the declaration of a member of the class with the same name; see 6.4.7. The declaration of a member in a derived class (11.7) hides the declaration of a member of a base class of the same name; see 11.8.

4 During the lookup of a name qualified by a namespace name, declarations that would otherwise be made visible by a using-directive can be hidden by declarations with the same name in the namespace containing the using-directive; see 6.5.4.3.

5 If a name is in scope and is not hidden it is said to be visible.

6.5 Name lookup [basic.lookup]

6.5.1 General [basic.lookup.general]

1 The name lookup rules apply uniformly to all names (including typedef-names (9.2.4), namespace-names (9.8), and class-names (11.3)) wherever the grammar allows such names in the context discussed by a particular rule. Name lookup associates the use of a name with a set of declarations (6.2) of that name. If the declarations found by name lookup all denote functions or function templates, the declarations are said to form an overload set. The declarations found by name lookup shall either all denote the same entity or form an overload set. Overload resolution (12.4, 12.5) takes place after name lookup has succeeded. The access rules (11.9) are considered only once name lookup and function overload resolution (if applicable) have succeeded. Only after name lookup, function overload resolution (if applicable) and access checking have succeeded are the semantic properties introduced by the name’s declaration and its reachable (10.7) redeclarations used further in expression processing (Clause 7).

2 A name “looked up in the context of an expression” is looked up in the scope where the expression is found.

3 The injected-class-name of a class (11.1) is also considered to be a member of that class for the purposes of name hiding and lookup.

[Note 1: 6.6 discusses linkage issues. The notions of scope, point of declaration and name hiding are discussed in 6.4. — end note]

6.5.2 Unqualified name lookup [basic.lookup.unqual]

1 In all the cases listed in 6.5.2, the scopes are searched for a declaration in the order listed in each of the respective categories; name lookup ends as soon as a declaration is found for the name. If no declaration is found, the program is ill-formed.
The declarations from the namespace nominated by a \textit{using-directive} become visible in a namespace enclosing the \textit{using-directive}; see 9.8.4. For the purpose of the unqualified name lookup rules described in 6.5.2, the declarations from the namespace nominated by the \textit{using-directive} are considered members of that enclosing namespace.

The lookup for an unqualified name used as the \textit{postfix-expression} of a function call is described in 6.5.3.

\[\text{Note 1}: \text{For purposes of determining (during parsing) whether an expression is a } \textit{postfix-expression} \text{ for a function call, the usual name lookup rules apply. In some cases a name followed by } < \text{ is treated as a } \textit{template-name} \text{ even though name lookup did not find a } \textit{template-name} \text{ (see 13.3). For example,}\]

\begin{verbatim}
int h;
void g();
namespace N {
    struct A {}
    template <class T> int f(T);
    template <class T> int g(T);
    template <class T> int h(T);
}

int x = f<N::A>());
// OK: lookup of \textit{f} finds nothing, \textit{f} treated as template name
int y = g<N::A>());
// OK: lookup of \textit{g} finds a function, \textit{g} treated as template name
int z = h<N::A>());
// error: \textit{h} does not begin a template-id
\end{verbatim}

The rules in 6.5.3 have no effect on the syntactic interpretation of an expression. For example,

\begin{verbatim}
typedef int f;
namespace N {
    struct A {
        friend void f(A &);
        operator int();
        void g(A a) {
            int i = f(a);
            // \textit{f} is the typedef, not the friend function: equivalent to int(a)
        }
    };
}

Because the expression is not a function call, the argument-dependent name lookup (6.5.3) does not apply and the friend function \textit{f} is not found. — end note\]

A name used in global scope, outside of any function, class or user-declared namespace, shall be declared before its use in global scope.

A name used in a user-declared namespace outside of the definition of any function or class shall be declared before its use in that namespace or before its use in a namespace enclosing its namespace.

In the definition of a function that is a member of namespace \textit{N}, a name used after the function’s \textit{declarator-id} shall be declared before its use in the block in which it is used or in one of its enclosing blocks (8.4) or shall be declared before its use in namespace \textit{N} or, if \textit{N} is a nested namespace, shall be declared before its use in one of \textit{N}’s enclosing namespaces.

\[\text{Example 1}:\]

\begin{verbatim}
namespace A {
    namespace N {
        void f();
    }
}

void A::N::f() {
    i = 5;
    // The following scopes are searched for a declaration of \textit{i}:
    // 1) outermost block scope of \textit{A::N::f}, before the use of \textit{i}
    // 2) scope of namespace \textit{N}
    // 3) scope of namespace \textit{A}
    // 4) global scope, before the definition of \textit{A::N::f}
}
\end{verbatim}

\[\text{Note 24}: \text{This refers to unqualified names that occur, for instance, in a type or default argument in the } \textit{parameter-declaration-clause} \text{ or used in the function body.}\]
A name used in the definition of a class \( X \) outside of a complete-class context \((11.4)\) of \( X \) shall be declared in one of the following ways:

1. before its use in class \( X \) or be a member of a base class of \( X \) \((11.8)\), or
2. if \( X \) is a nested class of class \( Y \) \((11.4.11)\), before the definition of \( X \) in \( Y \), or shall be a member of a base class of \( Y \) (this lookup applies in turn to \( Y \)'s enclosing classes, starting with the innermost enclosing class), \(^{26}\) or
3. if \( X \) is a local class \((11.6)\) or is a nested class of a local class, before the definition of class \( X \) in a block enclosing the definition of class \( X \), or
4. if \( X \) is a member of namespace \( N \), or is a nested class of a class that is a member of \( N \), or is a local class or a nested class within a local class of a function that is a member of \( N \), before the definition of class \( X \) in namespace \( N \) or in one of \( N \)'s enclosing namespaces.

### Example 2:
```cpp
namespace M {
    class B {
    }
}
namespace N {
    class Y : public M::B {
        class X {
            int a[1];
        };
    };
}
```

// The following scopes are searched for a declaration of \( i \):
// 1) scope of class N::Y::X, before the use of \( i \)
// 2) scope of class N::Y, before the definition of N::Y::X
// 3) scope of N::Y's base class M::B
// 4) scope of namespace N, before the definition of N::Y
// 5) global scope, before the definition of N

### Note 2: When looking for a prior declaration of a class or function introduced by a friend declaration, scopes outside of the innermost enclosing namespace scope are not considered; see 9.8.2.3. — end note]

### Note 3: 6.4.7 further describes the restrictions on the use of names in a class definition. 11.4.11 further describes the restrictions on the use of names in nested class definitions. 11.6 further describes the restrictions on the use of names in local class definitions. — end note]

For the members of a class \( X \), a name used in a complete-class context \((11.4)\) of \( X \) or in the definition of a class member outside of the definition of \( X \), following the member’s declarator-id \(^{27}\), shall be declared in one of the following ways:

1. before its use in the block in which it is used or in an enclosing block \((8.4)\), or
2. shall be a member of class \( X \) or be a member of a base class of \( X \) \((11.8)\), or
3. if \( X \) is a nested class of class \( Y \) \((11.4.11)\), shall be a member of \( Y \), or shall be a member of a base class of \( Y \) (this lookup applies in turn to \( Y \)'s enclosing classes, starting with the innermost enclosing class), \(^{26}\) or
4. if \( X \) is a local class \((11.6)\) or is a nested class of a local class, before the definition of class \( X \) in a block enclosing the definition of class \( X \), or
5. if \( X \) is a member of namespace \( N \), or is a nested class of a class that is a member of \( N \), or is a local class or a nested class within a local class of a function that is a member of \( N \), before the use of the name, in namespace \( N \) or in one of \( N \)'s enclosing namespaces.

\(^{25}\) This refers to unqualified names following the class name; such a name might be used in a base-specifier or in the member-specification of the class definition.

\(^{26}\) This lookup applies whether the definition of \( X \) is nested within \( Y \)'s definition or whether \( X \)'s definition appears in a namespace scope enclosing \( Y \)'s definition \((11.4.11)\).

\(^{27}\) That is, an unqualified name that occurs, for instance, in a type in the parameter-declaration-clause or in the noexcept-specifier.

\(^{28}\) This lookup applies whether the member function is defined within the definition of class \( X \) or whether the member function is defined in a namespace scope enclosing \( X \)'s definition.
Example 3:

```cpp
class B { };
namespace M {
    namespace N {
        class X : public B {
            void f();
        };
    }
}

void M::N::X::f() {
i = 16;
}
```

// The following scopes are searched for a declaration of i:
// 1) outermost block scope of M::N::X::f, before the use of i
// 2) scope of class M::N::X
// 3) scope of M::N::X's base class B
// 4) scope of namespace M::N
// 5) scope of namespace M
// 6) global scope, before the definition of M::N::X::f

—end example

Note 4: 11.4.2 and 11.4.9 further describe the restrictions on the use of names in member function definitions. 11.4.11 further describes the restrictions on the use of names in the scope of nested classes. 11.6 further describes the restrictions on the use of names in local class definitions. —end note

Name lookup for a name used in the definition of a friend function (11.9.4) defined inline in the class granting friendship shall proceed as described for lookup in member function definitions. If the friend function is not defined in the class granting friendship, name lookup in the friend function definition shall proceed as described for lookup in namespace member function definitions.

Example 4:

```cpp
struct A {
    typedef int AT;
    void f1(AT);
    void f2(float);
    template <class T> void f3();
};

struct B {
    typedef char AT;
    typedef float BT;
    friend void A::f1(AT); // parameter type is A::AT
    friend void A::f2(BT); // parameter type is B::BT
    friend void A::f3<AT>(); // template argument is B::AT
};
```

—end example

During the lookup for a name used as a default argument (9.3.4.7) in a function parameter-declaration-clause or used in the expression of a mem-initializer for a constructor (11.10.3), the function parameter names are visible and hide the names of entities declared in the block, class or namespace scopes containing the function declaration.

[Note 5: 9.3.4.7 further describes the restrictions on the use of names in default arguments. 11.10.3 further describes the restrictions on the use of names in a ctor-initializer. —end note]

During the lookup of a name used in the constant-expression of an enumerator-definition, previously declared enumerators of the enumeration are visible and hide the names of entities declared in the block, class, or namespace scopes containing the enum-specifier.

A name used in the definition of a static data member of class X (11.4.9.3) (after the qualified-id of the static member) is looked up as if the name was used in a member function of X.
If a variable member of a namespace is defined outside of the scope of its namespace then any name that appears in the definition of the member (after the declarator-id) is looked up as if the definition of the member occurred in its namespace.

Example 5:

```c
namespace N {
    int i = 4;
    extern int j;
}
int i = 2;
int N::j = i;  // N::j == 4
```

A name used in the handler for a function-try-block (14.1) is looked up as if the name was used in the outermost block of the function definition. In particular, the function parameter names shall not be redeclared in the exception-declaration nor in the outermost block of a handler for the function-try-block. Names declared in the outermost block of the function definition are not found when looked up in the scope of a handler for the function-try-block.

Note 7: But function parameter names are found. — end note

Note 8: The rules for name lookup in template definitions are described in 13.8. — end note

6.5.3 Argument-dependent name lookup

When the postfix-expression in a function call (7.6.1.3) is an unqualified-id, other namespaces not considered during the usual unqualified lookup (6.5.2) may be searched, and in those namespaces, namespace-scope friend function or function template declarations (11.9.4) not otherwise visible may be found. These modifications to the search depend on the types of the arguments (and for template template arguments, the namespace of the template argument).

Example 1:

```c
namespace N {
    struct S { };
    void f(S);
}

void g() {
    N::S s;
    f(S);  // OK: calls N::f
    (f)(s);  // error: N::f not considered; parentheses prevent argument-dependent lookup
}
```

For each argument type T in the function call, there is a set of zero or more associated namespaces and a set of zero or more associated entities (other than namespaces) to be considered. The sets of namespaces and entities are determined entirely by the types of the function arguments (and the namespace of any template template argument). Typedef names and using-declarations used to specify the types do not contribute to this set. The sets of namespaces and entities are determined in the following way:

(2.1) — If T is a fundamental type, its associated sets of namespaces and entities are both empty.

(2.2) — If T is a class type (including unions), its associated entities are: the class itself; the class of which it is a member, if any; and its direct and indirect base classes. Its associated namespaces are the innermost enclosing namespaces of its associated entities. Furthermore, if T is a class template specialization, its associated namespaces and entities also include: the namespaces and entities associated with the types of the template arguments provided for template type parameters (excluding template template parameters); the templates used as template template arguments; the namespaces of which any template template arguments are members; and the classes of which any member templates used as template template arguments are members.

Note 1: Non-type template arguments do not contribute to the set of associated namespaces. — end note
If \( T \) is an enumeration type, its associated namespace is the innermost enclosing namespace of its declaration, and its associated entities are \( T \) and, if it is a class member, the member’s class.

If \( T \) is a pointer to \( U \) or an array of \( U \), its associated namespaces and entities are those associated with \( U \).

If \( T \) is a function type, its associated namespaces and entities are those associated with the function parameter types and those associated with the return type.

If \( T \) is a pointer to a member function of a class \( X \), its associated namespaces and entities are those associated with the function parameter types and return type, together with those associated with \( X \).

If \( T \) is a pointer to a data member of class \( X \), its associated namespaces and entities are those associated with the member type together with those associated with \( X \).

If an associated namespace is an inline namespace (9.8.2), its enclosing namespace is also included in the set. If an associated namespace directly contains inline namespaces, those inline namespaces are also included in the set. In addition, if the argument is the name or address of an overload set, its associated entities and namespaces are the union of those associated with each of the members of the set, i.e., the entities and namespaces associated with its parameter types and return type. Additionally, if the aforementioned overload set is named with a template-id, its associated entities and namespaces also include those of its type template-arguments and its template template-arguments.

Let \( X \) be the lookup set produced by unqualified lookup (6.5.2) and let \( Y \) be the lookup set produced by argument dependent lookup (defined as follows). If \( X \) contains

- a declaration of a class member, or
- a block-scope function declaration that is not a using-declaration, or
- a declaration that is neither a function nor a function template

then \( Y \) is empty. Otherwise \( Y \) is the set of declarations found in the namespaces associated with the argument types as described below. The set of declarations found by the lookup of the name is the union of \( X \) and \( Y \).

[Note 2: The namespaces and entities associated with the argument types can include namespaces and entities already considered by the ordinary unqualified lookup. — end note]

[Example 2: The namespaces and entities associated with the argument types can include namespaces and entities already considered by the ordinary unqualified lookup. — end note]

When considering an associated namespace \( N \), the lookup is the same as the lookup performed when \( N \) is used as a qualifier (6.5.4.3) except that:

- Any using-directives in \( N \) are ignored.
- All names except those of (possibly overloaded) functions and function templates are ignored.
- Any namespace-scope friend functions or friend function templates (11.9.4) declared in classes with reachable definitions in the set of associated entities are visible within their respective namespaces even if they are not visible during an ordinary lookup (9.8.2.3).
- Any exported declaration \( D \) in \( N \) declared within the purview of a named module \( M \) (10.2) is visible if there is an associated entity attached to \( M \) with the same innermost enclosing non-inline namespace as \( D \).
- If the lookup is for a dependent name (13.8.3, 13.8.5.2), any declaration \( D \) in \( N \) is visible if \( D \) would be visible to qualified name lookup (6.5.4.3) at any point in the instantiation context (10.6) of the
lookup, unless D is declared in another translation unit, attached to the global module, and is either discarded (10.4) or has internal linkage.

[Example 3:

Translation unit #1:

```cpp
export module M;
namespace R {
  export struct X {};
  export void f(X);
}
namespace S {
  export void f(R::X, R::X);
}
```

Translation unit #2:

```cpp
export module N;
import M;
export R::X make();
namespace R { static int g(X); }
export template<typename T, typename U> void apply(T t, U u) {
  f(t, u);
  g(t);
}
```

Translation unit #3:

```cpp
module Q;
import N;
namespace S {
  struct Z { template<typename T> operator T(); }
}
void test() {
  auto x = make(); // OK, decltype(x) is R::X in module M
  R::f(x); // error: R and R::f are not visible here
  f(x); // OK, calls R::f from interface of M
  f(x, S::Z()); // error: S::f in module M not considered
  // even though S is an associated namespace
  apply(x, S::Z()); // error: S::f is visible in instantiation context, but
  // R::g has internal linkage and cannot be used outside TU #2
}
```
—end example]

6.5.4 Qualified name lookup

6.5.4.1 General

The name of a class or namespace member or enumerator can be referred to after the :: scope resolution operator (7.5.4.3) applied to a nested-name-specifier that denotes its class, namespace, or enumeration. If a :: scope resolution operator in a nested-name-specifier is not preceded by a decltype-specifier, lookup of the name preceding that :: considers only namespaces, types, and templates whose specializations are types. If the name found does not designate a namespace or a class, enumeration, or dependent type, the program is ill-formed.

[Example 1:

```cpp
class A {
  public:
    static int n;
};
int main() {
  int A;
  A::n = 42; // OK
  A b; // error: A does not name a type
}
```
—end example]
[Note 1: Multiply qualified names, such as N1::N2::N3::n, can be used to refer to members of nested classes (11.4.11) or members of nested namespaces. — end note]

In a declaration in which the declarator-id is a qualified-id, names used before the qualified-id being declared are looked up in the defining namespace scope; names following the qualified-id are looked up in the scope of the member’s class or namespace.

[Example 2:

class X {};

class C {
    class X {};
    static const int number = 50;
    static X arr[number];
};

X C::arr[number]; // error:
    // equivalent to ::X C::arr[C::number];
    // and not to C::X C::arr[C::number];

— end example]

A name prefixed by the unary scope operator :: (7.5.4.3) is looked up in global scope, in the translation unit where it is used. The name shall be declared in global namespace scope or shall be a name whose declaration is visible in global scope because of a using-directive (6.5.4.3). The use of :: allows a global name to be referred to even if its identifier has been hidden (6.4.10).

A name prefixed by a nested-name-specifier that nominates an enumeration type shall represent an enumerator of that enumeration.

In a qualified-id of the form:

\[ \text{nested-name-specifier}_\text{opt} \text{ type-name} :: \sim \text{type-name} \]

the second type-name is looked up in the same scope as the first.

[Example 3:

struct C {
    typedef int I;
};

typedef int I1, I2;
extern int* p;
extern int* q;
p->C::I::~C(); // I is looked up in the scope of C
q->I1::~I2(); // I2 is looked up in the scope of the postfix-expression

struct A {
    ~A();
};
typedef A AB;
int main() {
    AB* p;
p->AB::~AB(); // explicitly calls the destructor for A
}

— end example]

[Note 2: 6.5.6 describes how name lookup proceeds after the . and \text{-} operators. — end note]

6.5.4.2 Class members [class.qual]

If the nested-name-specifier of a qualified-id nominates a class, the name specified after the nested-name-specifier is looked up in the scope of the class (11.8), except for the cases listed below. The name shall represent one or more members of that class or of one of its base classes (11.7).

[Note 1: A class member can be referred to using a qualified-id at any point in its potential scope (6.4.7). — end note]

The exceptions to the name lookup rule above are the following:

1. the lookup for a destructor is as specified in 6.5.4;
2. a conversion-type-id of a conversion-function-id is looked up in the same manner as a conversion-type-id in a class member access (see 6.5.6);
the names in a template-argument of a template-id are looked up in the context in which the entire
postfix-expression occurs;

— the lookup for a name specified in a using-declaration (9.9) also finds class or enumeration names hidden
within the same scope (6.4.10).

(1.4) In a lookup in which function names are not ignored and the nested-name-specifier nominates a class C:

— if the name specified after the nested-name-specifier, when looked up in C, is the injected-class-name of
C (11.1), or

— in a using-declarator of a using-declaration (9.9) that is a member-declaration, if the name specified after
the nested-name-specifier is the same as the identifier or the simple-template-id’s template-name in the
last component of the nested-name-specifier,

the name is instead considered to name the constructor of class C.

[Note 2: For example, the constructor is not an acceptable lookup result in an elaborated-type-specifier so the
constructor would not be used in place of the injected-class-name. — end note]

Such a constructor name shall be used only in the declarator-id of a declaration that names a constructor or in
a using-declaration.

[Example 1:

```c
struct A { A();
};
struct B: public A { B();
};

A::A() { }
B::B() { }

B::A ba;           // object of type A
A::A a;           // error: A::A is not a type name
struct A::A a2;    // object of type A
```

— end example]

3 A class member name hidden by a name in a nested declarative region or by the name of a derived class
member can still be found if qualified by the name of its class followed by the :: operator.

6.5.4.3 Namespace members [namespace.qual]

1 If the nested-name-specifier of a qualified-id nominates a namespace (including the case where the nested-
name-specifier is ::, i.e., nominating the global namespace), the name specified after the nested-name-specifier
is looked up in the scope of the namespace. The names in a template-argument of a template-id are looked up
in the context in which the entire postfix-expression occurs.

2 For a namespace X and name m, the namespace-qualified lookup set S(X, m) is defined as follows: Let
S′(X, m) be the set of all declarations of m in X and the inline namespace set of X (9.8.2) whose potential
scope (6.4.6) would include the namespace in which m is declared at the location of the nested-name-specifier.
If S′(X, m) is not empty, S(X, m) is S′(X, m); otherwise, S(X, m) is the union of S(Ni, m) for all namespaces
Ni nominated by using-directives in X and its inline namespace set.

3 Given X::m (where X is a user-declared namespace), or given ::m (where X is the global namespace), if
S(X, m) is the empty set, the program is ill-formed. Otherwise, if S(X, m) has exactly one member, or if
the context of the reference is a using-declaration (9.9), S(X, m) is the required set of declarations of m.
Otherwise if the use of m is not one that allows a unique declaration to be chosen from S(X, m), the program
is ill-formed.

[Example 1:

```c
int x;
namespace Y {
    void f(float);
    void h(int);
}
```

29) Lookups in which function names are ignored include names appearing in a nested-name-specifier, an elaborated-type-specifier, or a base-specifier.
namespace Z {
    void h(double);
}

namespace A {
    using namespace Y;
    void f(int);
    void g(int);
    int i;
}

namespace B {
    using namespace Z;
    void f(char);
    int i;
}

namespace AB {
    using namespace A;
    using namespace B;
    void g();
}

void h() {
    AB::g(); // g is declared directly in AB, therefore S is {AB::g()} and AB::g() is chosen
    AB::f(1); // f is not declared directly in AB so the rules are applied recursively to A and B;
               // namespace Y is not searched and Y::f(float) is not considered;
               // S is {A::f(int),B::f(char)} and overload resolution chooses A::f(int)
    AB::f('c'); // as above but resolution chooses B::f(char)
    AB::x++; // x is not declared directly in AB, and is not declared in A or B, so the rules
               // are applied recursively to Y and Z, S is {} so the program is ill-formed
    AB::i++; // i is not declared directly in AB so the rules are applied recursively to A and B,
               // S is {A::i,B::i} so the use is ambiguous and the program is ill-formed
    AB::h(16.8); // h is not declared directly in AB and not declared directly in A or B so the rules
                  // are applied recursively to Y and Z, S is {Y::h(int),Z::h(double)} and
                  // overload resolution chooses Z::h(double)
}

—end example] 4

[Note 1: The same declaration found more than once is not an ambiguity (because it is still a unique declaration).

Example 2:

namespace A {
    int a;
}

namespace B {
    using namespace A;
}

namespace C {
    using namespace A;
}

namespace BC {
    using namespace B;
    using namespace C;
}
void f()
{
    BC::a++;
    // OK: S is {A::a,A::a}
}

namespace D {
    using A::a;
}

namespace BD {
    using namespace B;
    using namespace D;
}

void g()
{
    BD::a++;
    // OK: S is {A::a,A::a}
}
—end example]
—end note]

[Example 3: Because each referenced namespace is searched at most once, the following is well-defined:

namespace B {
    int b;
}

namespace A {
    using namespace B;
    int a;
}

namespace B {
    using namespace A;
}

void f()
{
    A::a++;
    // OK: a declared directly in A, S is {A::a}
    B::a++;
    // OK: both A and B searched (once), S is {A::a}
    A::b++;
    // OK: both A and B searched (once), S is {B::b}
    B::b++;
    // OK: b declared directly in B, S is {B::b}
}
—end example]

During the lookup of a qualified namespace member name, if the lookup finds more than one declaration of
the member, and if one declaration introduces a class name or enumeration name and the other declarations
introduce either the same variable, the same enumerator, or a set of functions, the non-type name hides
the class or enumeration name if and only if the declarations are from the same namespace; otherwise (the
declarations are from different namespaces), the program is ill-formed.

[Example 4:

namespace A {
    struct x { };  
    int x;    
    int y;
}

namespace B {
    struct y { };  
}

namespace C {
    using namespace A;
}
using namespace B;
int i = C::x; // OK, A::x (of type int)
int j = C::y; // ambiguous, A::y or B::y

—end example]

In a declaration for a namespace member in which the declarator-id is a qualified-id, given that the qualified-id for the namespace member has the form

\[\text{nested-name-specifier unqualified-id}\]

the unqualified-id shall name a member of the namespace designated by the nested-name-specifier or of an element of the inline namespace set (9.8.2) of that namespace.

[Example 5:

```
namespace A {
    namespace B {
        void f1(int);
    }
    using namespace B;
}
void A::f1(int){ } // error: f1 is not a member of A
```
—end example]

However, in such namespace member declarations, the nested-name-specifier may rely on using-directives to implicitly provide the initial part of the nested-name-specifier.

[Example 6:

```
namespace A {
    namespace B {
        void f1(int);
    }
}
namespace C {
    namespace D {
        void f1(int);
    }
}
using namespace A;
using namespace C::D;
void B::f1(int){ } // OK, defines A::B::f1(int)
```
—end example]

6.5.5 Elaborated type specifiers

An elaborated-type-specifier (9.2.9.4) may be used to refer to a previously declared class-name or enum-name even though the name has been hidden by a non-type declaration (6.4.10).

If the elaborated-type-specifier has no nested-name-specifier, and unless the elaborated-type-specifier appears in a declaration with the following form:

\[\text{class-key attribute-specifier-seqopt identifier }\]

the identifier is looked up according to 6.5.2 but ignoring any non-type names that have been declared. If the elaborated-type-specifier is introduced by the enum keyword and this lookup does not find a previously declared type-name, the elaborated-type-specifier is ill-formed. If the elaborated-type-specifier is introduced by the class-key and this lookup does not find a previously declared type-name, or if the elaborated-type-specifier appears in a declaration with the form:

\[\text{class-key attribute-specifier-seqopt identifier }\]

the elaborated-type-specifier is a declaration that introduces the class-name as described in 6.4.2.

If the elaborated-type-specifier has a nested-name-specifier, qualified name lookup is performed, as described in 6.5.4, but ignoring any non-type names that have been declared. If the name lookup does not find a previously declared type-name, the elaborated-type-specifier is ill-formed.
Example 1:

```c
struct Node {
    struct Node* Next;   // OK: Refers to injected-class-name Node
    struct Data* Data;   // OK: Declares type Data at global scope and member Data
};

struct Data {
    struct Node* Node;    // OK: Refers to Node at global scope
    friend struct ::Glob; // error: Glob is not declared, cannot introduce a qualified type (9.2.9.4)
    friend struct Glob;   // OK: Refers to (as yet) undeclared Glob at global scope.
    /* ... */
};

struct Base {
    struct Data;          // OK: Declares nested Data
    struct ::Data* thatData; // OK: Refers to ::Data
    struct Base::Data* thisData; // OK: Refers to nested Data
    friend class ::Data;  // OK: global Data is a friend
    friend class Data;    // OK: nested Data is a friend
    struct Data { /* ... */ }; // Defines nested Data
};

struct Data;                   // OK: Redeclares Data at global scope
struct ::Data;                 // error: cannot introduce a qualified type (9.2.9.4)
struct Base::Data;             // error: cannot introduce a qualified type (9.2.9.4)
struct Base::Datum;            // error: Datum undefined
struct Base::Data* pBase;      // OK: refers to nested Data
```

—end example—

### 6.5.6 Class member access

In a class member access expression (7.6.1.5), if the . or -> token is immediately followed by an identifier followed by a <, the identifier is looked up to determine whether the < is the beginning of a template argument list (13.3) or a less-than operator. The identifier is first looked up in the class of the object expression (11.8). If the identifier is not found, it is then looked up in the context of the entire postfix-expression and shall name a template whose specializations are types.

1. If the id-expression in a class member access (7.6.1.5) is an unqualified-id, and the type of the object expression is of a class type C, the unqualified-id is looked up in the scope of class C (11.8).

2. If the unqualified-id is -type-name, the type-name is looked up in the context of the entire postfix-expression. If the type T of the object expression is of a class type C, the type-name is also looked up in the scope of class C. At least one of the lookups shall find a name that refers to cv T.

Example 1:

```c
struct A { }

struct B {
    void f(::A* a);  
};

void B::f(::A* a) {
    a->~A();       // OK: lookup in *a finds the injected-class-name
}
```

—end example—

4. If the id-expression in a class member access is a qualified-id of the form

```
class-name-or-namespace-name::...
```

the class-name-or-namespace-name following the . or -> operator is first looked up in the class of the object expression (11.8) and the name, if found, is used. Otherwise it is looked up in the context of the entire postfix-expression.
[Note 1: See 6.5.4, which describes the lookup of a name before ::, which will only find a type or namespace name. — end note]

5 If the qualified-id has the form

```
::class-name-or-namespace-name::...
```

the class-name-or-namespace-name is looked up in global scope as a class-name or namespace-name.

6 If the nested-name-specifier contains a simple-template-id (13.3), the names in its template-arguments are looked up in the context in which the entire postfix-expression occurs.

7 If the id-expression is a conversion-function-id, its conversion-type-id is first looked up in the class of the object expression (11.8) and the name, if found, is used. Otherwise it is looked up in the context of the entire postfix-expression. In each of these lookups, only names that denote types or templates whose specializations are types are considered.

[Example 2:]

```cpp
struct A { }
namespace N {
  struct A {
    void g() { }
    template <class T> operator T();
  };
}

int main() {
  N::A a;
  a.operator A(); // calls N::A::operator N::A
}
```

— end example]

### 6.5.7 Using-directives and namespace aliases

In a using-directive or namespace-alias-definition, during the lookup for a namespace-name or for a name in a nested-name-specifier only namespace names are considered.

### 6.6 Program and linkage

A program consists of one or more translation units (5.1) linked together. A translation unit consists of a sequence of declarations.

[translation-unit:]

```
declaration-seqopt
global-module-fragmentopt module-declaration declaration-seqopt private-module-fragmentopt
```

2 A name is said to have linkage when it can denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

1. When a name has external linkage, the entity it denotes can be referred to by names from scopes of other translation units or from other scopes of the same translation unit.
2. When a name has module linkage, the entity it denotes can be referred to by names from other scopes of the same module unit (10.1) or from scopes of other module units of that same module.
3. When a name has internal linkage, the entity it denotes can be referred to by names from other scopes in the same translation unit.
4. When a name has no linkage, the entity it denotes cannot be referred to by names from other scopes.

3 A name having namespace scope (6.4.6) has internal linkage if it is the name of

1. a variable, variable template, function, or function template that is explicitly declared static; or
2. a non-template variable of non-volatile const-qualified type, unless
   1. it is explicitly declared extern, or
   2. it is inline or exported, or
   3. it was previously declared and the prior declaration did not have internal linkage; or
   4. a data member of an anonymous union.

§ 6.6
An unnamed namespace or a namespace declared directly or indirectly within an unnamed namespace has internal linkage. All other namespaces have external linkage. A name having namespace scope that has not been given internal linkage above and that is the name of

- a variable; or
- a function; or
- a named class (11.1), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (9.2.4); or
- a named enumeration (9.7.1), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (9.2.4); or
- an unnamed enumeration that has an enumerator as a name for linkage purposes (9.7.1); or
- a template

has its linkage determined as follows:

- if the enclosing namespace has internal linkage, the name has internal linkage;
- otherwise, if the declaration of the name is attached to a named module (10.1) and is not exported (10.2), the name has module linkage;
- otherwise, the name has external linkage.

In addition, a member function, static data member, a named class or enumeration of class scope, or an unnamed class or enumeration defined in a class-scope typedef declaration such that the class or enumeration has the typedef name for linkage purposes (9.2.4), has the same linkage, if any, as the name of the class of which it is a member.

The name of a function declared in block scope and the name of a variable declared by a block scope `extern` declaration have linkage. If such a declaration is attached to a named module, the program is ill-formed. If there is a visible declaration of an entity with linkage, ignoring entities declared outside the innermost enclosing namespace scope, such that the block scope declaration would be a (possibly ill-formed) redeclaration if the two declarations appeared in the same declarative region, the block scope declaration declares that same entity and receives the linkage of the previous declaration. If there is more than one such matching entity, the program is ill-formed. Otherwise, if no matching entity is found, the block scope entity receives external linkage. If, within a translation unit, the same entity is declared with both internal and external linkage, the program is ill-formed.

**Example 1:**

```c
static void f();
extern "C" void h();
static int i = 0;          // #1
void g() {
  extern void f();  // internal linkage
  extern void h();  // C language linkage
  int i;            // #2: i has no linkage
  {
    extern void f(); // internal linkage
    extern int i;    // #3: external linkage, ill-formed
  }
}
```

Without the declaration at line #2, the declaration at line #3 would link with the declaration at line #1. Because the declaration with internal linkage is hidden, however, #3 is given external linkage, making the program ill-formed. [end example]

When a block scope declaration of an entity with linkage is not found to refer to some other declaration, then that entity is a member of the innermost enclosing namespace. However such a declaration does not introduce the member name in its namespace scope.

**Example 2:**

```c
namespace X { 
  void p() {
```
8 Names not covered by these rules have no linkage. Moreover, except as noted, a name declared at block scope (6.4.3) has no linkage.

9 Two names that are the same (6.1) and that are declared in different scopes shall denote the same variable, function, type, template or namespace if

(9.1) — both names have external or module linkage and are declared in declarations attached to the same module, or else both names have internal linkage and are declared in the same translation unit; and

(9.2) — both names refer to members of the same namespace or to members, not by inheritance, of the same class; and

(9.3) — when both names denote functions or function templates, the signatures (3.51, 3.53) are the same.

If multiple declarations of the same name with external linkage would declare the same entity except that they are attached to different modules, the program is ill-formed; no diagnostic is required.

[Note 2: using-declarations, typedef declarations, and alias-declarations do not declare entities, but merely introduce synonyms. Similarly, using-directives do not declare entities. Enumerators do not have linkage, but might serve as the name of an enumeration with linkage (9.7.1). — end note]

10 If a declaration would redeclare a reachable declaration attached to a different module, the program is ill-formed.

[Example 3:

"decls.h":
int f(); // #1, attached to the global module
int g(); // #2, attached to the global module

Module interface of M:
module;
#include "decls.h"
export module M;
export using ::f; // OK: does not declare an entity, exports #1
int g(); // error: matches #2, but attached to M
export int h(); // #3
export int k(); // #4

Other translation unit:
import M;
static int h(); // error: matches #3
int k(); // error: matches #4

— end example]

As a consequence of these rules, all declarations of an entity are attached to the same module; the entity is said to be attached to that module.

11 After all adjustments of types (during which typedefs (9.2.4) are replaced by their definitions), the types specified by all declarations referring to a given variable or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (9.3.4.5). A violation of this rule on type identity does not require a diagnostic.

[Note 3: Linkage to non-C++ declarations can be achieved using a linkage-specification (9.11). — end note]
A declaration $D$ names an entity $E$ if

- $D$ contains a lambda-expression whose closure type is $E$,
- $E$ is not a function or function template and $D$ contains an id-expression, type-specifier, nested-name-specifier, template-name, or concept-name denoting $E$, or
- $E$ is a function or function template and $D$ contains an expression that names $E$ (6.3) or an id-expression that refers to a set of overloads that contains $E$.

[Note 4: Non-dependent names in an instantiated declaration do not refer to a set of overloads (13.8.4). — end note]

A declaration is an exposure if it either names a TU-local entity (defined below), ignoring

- the function-body for a non-inline function or function template (but not the deduced return type for a (possibly instantiated) definition of a function with a declared return type that uses a placeholder type (9.2.9.6)),
- the initializer for a variable or variable template (but not the variable’s type),
- friend declarations in a class definition, and
- any reference to a non-volatile const object or reference with internal or no linkage initialized with a constant expression that is not an odr-use (6.3), or defines a constexpr variable initialized to a TU-local value (defined below).

[Note 5: An inline function template can be an exposure even though explicit specializations of it might be usable in other translation units. — end note]

An entity is TU-local if it is

- a type, function, variable, or template that
  - has a name with internal linkage, or
  - does not have a name with linkage and is declared, or introduced by a lambda-expression, within the definition of a TU-local entity,
- a type with no name that is defined outside a class-specifier, function body, or initializer or is introduced by a defining-type-specifier that is used to declare only TU-local entities,
- a specialization of a TU-local template,
- a specialization of a template with any TU-local template argument, or
- a specialization of a template whose (possibly instantiated) declaration is an exposure.

[Note 6: The specialization might have been implicitly or explicitly instantiated. — end note]

A value or object is TU-local if either

- it is, or is a pointer to, a TU-local function or the object associated with a TU-local variable, or
- it is an object of class or array type and any of its subobjects or any of the objects or functions to which its non-static data members of reference type refer is TU-local and is usable in constant expressions.

If a (possibly instantiated) declaration of, or a deduction guide for, a non-TU-local entity in a module interface unit (outside the private-module-fragment, if any) or module partition (10.1) is an exposure, the program is ill-formed. Such a declaration in any other context is deprecated (D.8).

If a declaration that appears in one translation unit names a TU-local entity declared in another translation unit that is not a header unit, the program is ill-formed. A declaration instantiated for a template specialization (13.9) appears at the point of instantiation of the specialization (13.8.5.1).

[Example 4:]
```cpp
decltype(f) *fp; // error: f (though not its type) is TU-local
auto &fr = f; // OK
constexpr auto &fr2 = f; // error: is an exposure of f
constexpr static auto fp2 = fr; // OK
struct S { void (&ref)(); } s(f); // OK, value is TU-local
constexpr extern struct W { S &s; } wrap{s}; // OK, value is not TU-local

static auto x = []{f();}; // OK
auto x2 = x; // error: the closure type is TU-local
int y = ([]{f();},0); // error: the closure type is not TU-local
int y2 = (x,0); // OK
namespace N {
    struct A {}; 
    void adl(A);
    static void adl(int);
}
void adl(double);

inline void h(auto x) { adl(x); } // OK, but a specialization might be an exposure

Translation unit #2:

module A;
void other() { 
g<0>(); // OK, specialization is explicitly instantiated
  g<1>(); // error: specialization uses TU-local its
  h(N::A{})); // error: overload set contains TU-local N::adl(int)
  h(0); // OK, calls adl(double)
  adl(N::A{})); // OK; N::adl(int) not found, calls N::adl(N::A)
  fr(); // OK, calls f
  constexpr auto ptr = fr; // error: fr is not usable in constant expressions here
}

—end example]```

6.7 Memory and objects

6.7.1 Memory model

The fundamental storage unit in the C++ memory model is the **byte**. A byte is at least large enough to contain any member of the basic execution character set (5.3) and the eight-bit code units of the Unicode® UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the **low-order bit**; the most significant bit is called the **high-order bit**. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

A **memory location** is either an object of scalar type that is not a bit-field or a maximal sequence of adjacent bit-fields all having nonzero width.

Two or more threads of execution (6.9.2) can access separate memory locations without interfering with each other.

---

[Note 1: The representation of types is described in 6.8. — end note]

[Note 2: Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. — end note]

[Note 3: Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of nonzero width. — end note]

[Example 1: A class declared as

---

30) Unicode® is a registered trademark of Unicode, Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

31) The number of bits in a byte is reported by the macro `CHAR_BIT` in the header `<climits>` (17.3.6).
contains four separate memory locations: The member \texttt{a} and bit-fields \texttt{d} and \texttt{e.ee} are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields \texttt{b} and \texttt{c} together constitute the fourth memory location. The bit-fields \texttt{b} and \texttt{c} cannot be concurrently modified, but \texttt{b} and \texttt{a}, for example, can be.

--- end example ---

6.7.2 Object model

The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An \textit{object} is created by a definition (6.2), by a \textit{new-expression} (7.6.2.8), by an operation that implicitly creates objects (see below), when implicitly changing the active member of a union (11.5), or when a temporary object is created (7.3.5, 6.7.7). An object occupies a region of storage in its period of construction (11.10.5), throughout its lifetime (6.7.3), and in its period of destruction (11.10.5).

[Note 1: A function is not an object, regardless of whether or not it occupies storage in the way that objects do. —end note]

The properties of an object are determined when the object is created. An object can have a name (6.1). An object has a storage duration (6.7.5) which influences its lifetime (6.7.3). An object has a type (6.8). Some objects are polymorphic (11.7.3); the implementation generates information associated with each such object that makes it possible to determine that object’s type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the \textit{expressions} (7.6) used to access them.

2 Objects can contain other objects, called \textit{subobjects}. A subobject can be a \textit{member subobject} (11.4), a \textit{base class subobject} (11.7), or an array element. An object that is not a subobject of any other object is called a \textit{complete object}. If an object is created in storage associated with a member subobject or array element \texttt{e} (which may or may not be within its lifetime), the created object is a subobject of \texttt{e}’s containing object if:

\begin{enumerate}
  \item[(2.1)] the lifetime of \texttt{e}’s containing object has begun and not ended, and
  \item[(2.2)] the storage for the new object exactly overlays the storage location associated with \texttt{e}, and
  \item[(2.3)] the new object is of the same type as \texttt{e} (ignoring cv-qualification).
\end{enumerate}

3 If a complete object is created (7.6.2.8) in storage associated with another object \texttt{e} of type “array of \texttt{N unsigned char}” or of type “array of \texttt{N std::byte}” (17.2.1), that array \textit{provides storage} for the created object if:

\begin{enumerate}
  \item[(3.1)] the lifetime of \texttt{e} has begun and not ended, and
  \item[(3.2)] the storage for the new object fits entirely within \texttt{e}, and
  \item[(3.3)] there is no smaller array object that satisfies these constraints.
\end{enumerate}

[Note 2: If that portion of the array previously provided storage for another object, the lifetime of that object ends because its storage was reused (6.7.3). —end note]

[Example 1:]

```cpp
#include <cstdint>

template<typename ...T>
struct AlignedUnion {
    alignas(T...) unsigned char data[max(sizeof(T)...)];
};

int f() {
    AlignedUnion<int, char> au;
    int *p = new (au.data) int; // OK, au.data provides storage
    char *c = new (au.data) char(); // OK, ends lifetime of *p
    char *d = new (au.data + 1) char(); // OK
    return *c + *d;
}
```

§ 6.7.2 59
struct A { unsigned char a[32]; };  
struct B { unsigned char b[16]; };  
A a;  
B *b = new (a.a + 8) B;  
// a.a provides storage for *b  
int *p = new (b->b + 4) int;  
// b->b provides storage for *p  
// a.a does not provide storage for *p (directly).  
// but *p is nested within a (see below)  

--- end example ---

4 An object a is nested within another object b if:

(4.1) — a is a subobject of b, or
(4.2) — b provides storage for a, or
(4.3) — there exists an object c where a is nested within c, and c is nested within b.

5 For every object x, there is some object called the complete object of x, determined as follows:

(5.1) — If x is a complete object, then the complete object of x is itself.
(5.2) — Otherwise, the complete object of x is the complete object of the (unique) object that contains x.

6 If a complete object, a member subobject, or an array element is of class type, its type is considered the most derived class, to distinguish it from the class type of any base class subobject; an object of a most derived class type or of a non-class type is called a most derived object.

7 A potentially-overlapping subobject is either:

(7.1) — a base class subobject, or
(7.2) — a non-static data member declared with the no_unique_address attribute (9.12.10).

8 An object has nonzero size if it

(8.1) — is not a potentially-overlapping subobject, or
(8.2) — is not of class type, or
(8.3) — is of a class type with virtual member functions or virtual base classes, or
(8.4) — has subobjects of nonzero size or unnamed bit-fields of nonzero length.

Otherwise, if the object is a base class subobject of a standard-layout class type with no non-static data members, it has zero size. Otherwise, the circumstances under which the object has zero size are implementation-defined. Unless it is a bit-field (11.4.10), an object with nonzero size shall occupy one or more bytes of storage, including every byte that is occupied in full or in part by any of its subobjects. An object of trivially copyable or standard-layout type (6.8) shall occupy contiguous bytes of storage.

9 Unless an object is a bit-field or a subobject of zero size, the address of that object is the address of the first byte it occupies. Two objects with overlapping lifetimes that are not bit-fields may have the same address if one is nested within the other, or if at least one is a subobject of zero size and they are of different types; otherwise, they have distinct addresses and occupy disjoint bytes of storage.

[Example 2]:

    static const char test1 = 'x';
    static const char test2 = 'x';
    const bool b = &test1 != &test2;  // always true

--- end example ---

The address of a non-bit-field subobject of zero size is the address of an unspecified byte of storage occupied by the complete object of that subobject.

10 Some operations are described as implicitly creating objects within a specified region of storage. For each operation that is specified as implicitly creating objects, that operation implicitly creates and starts the lifetime of zero or more objects of implicit-lifetime types (6.8) in its specified region of storage if doing so would result in the program having defined behavior. If no such set of objects would give the program defined behavior, the behavior of the program is undefined. If multiple such sets of objects would give the program defined behavior, it is unspecified which such set of objects is created.

32) Under the “as-if” rule an implementation is allowed to store two objects at the same machine address or not store an object at all if the program cannot observe the difference (6.9.1).
Further, after implicitly creating objects within a specified region of storage, some operations are described as producing a pointer to a suitable created object. These operations select one of the implicitly-created objects whose address is the address of the start of the region of storage, and produce a pointer value that points to that object, if that value would result in the program having defined behavior. If no such pointer value would give the program defined behavior, the behavior of the program is undefined. If multiple such pointer values would give the program defined behavior, it is unspecified which such pointer value is produced.

Example 3:

```cpp
#include <cstdlib>
struct X { int a, b; }
X *make_x() {
    // The call to std::malloc implicitly creates an object of type X
    // and its subobjects a and b, and returns a pointer to that X object
    // (or an object that is pointer-interconvertible (6.8.3) with it),
    // in order to give the subsequent class member access operations
    // defined behavior.
    X *p = (X*)std::malloc(sizeof(struct X));
    p->a = 1;
    p->b = 2;
    return p;
}
```

An operation that begins the lifetime of an array of char, unsigned char, or std::byte implicitly creates objects within the region of storage occupied by the array.

Note 4: The array object provides storage for these objects. — end note]

Any implicit or explicit invocation of a function named operator new or operator new[] implicitly creates objects in the returned region of storage and returns a pointer to a suitable created object.

Note 5: Some functions in the C++ standard library implicitly create objects (20.10.9.3, 20.10.12, 21.5.3, 26.5.3). — end note]

6.7.3 Lifetime [basic.life]

The lifetime of an object or reference is a runtime property of the object or reference. A variable is said to have vacuous initialization if it is default-initialized and, if it is of class type or a (possibly multi-dimensional) array thereof, that class type has a trivial default constructor. The lifetime of an object of type T begins when:

1. storage with the proper alignment and size for type T is obtained, and
2. its initialization (if any) is complete (including vacuous initialization) (9.4),

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (9.4.2, 11.10.3), or as described in 11.5 and 11.4.5.3, and except as described in 20.10.10.2. The lifetime of an object o of type T ends when:

1. if T is a non-class type, the object is destroyed, or
2. if T is a class type, the destructor call starts, or
3. the storage which the object occupies is released, or is reused by an object that is not nested within o (6.7.2).

The lifetime of a reference begins when its initialization is complete. The lifetime of a reference ends as if it were a scalar object requiring storage.

Note 1: 11.10.3 describes the lifetime of base and member subobjects. — end note]

The properties ascribed to objects and references throughout this document apply for a given object or reference only during its lifetime.

Note 2: In particular, before the lifetime of an object starts and after its lifetime ends there are significant restrictions on the use of the object, as described below, in 11.10.3 and in 11.10.5. Also, the behavior of an object under construction and destruction might not be the same as the behavior of an object whose lifetime has started and not...
A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling a destructor or pseudo-destructor (7.5.4.4) for the object. For an object of a class type, the program is not required to call the destructor explicitly before the storage which the object occupies is reused or released; however, if there is no explicit call to the destructor or if a `delete-expression` (7.6.2.9) is not used to release the storage, the destructor is not implicitly called and any program that depends on the side effects produced by the destructor has undefined behavior.

Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that represents the address of the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see 11.10.5. Otherwise, such a pointer refers to allocated storage (6.7.5.5.2), and using the pointer as if the pointer were of type `void*` is well-defined. Indirection through such a pointer is permitted but the resulting value may only be used in limited ways, as described below. The program has undefined behavior if:

1. the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a `delete-expression`,
2. the pointer is used to access a non-static data member or call a non-static member function of the object, or
3. the pointer is implicitly converted (7.3.12) to a pointer to a virtual base class, or
4. the pointer is used as the operand of a `static_cast` (7.6.1.9), except when the conversion is to `cv void`, or to pointer to `cv void` and subsequently to pointer to `cv char`, `cv unsigned char`, or `cv std::byte` (17.2.1), or
5. the pointer is used as the operand of a `dynamic_cast` (7.6.1.7).

Example 1:

```cpp
#include <cstdlib>

struct B {
    virtual void f();
    void mutate();
    virtual ~B();
};

struct D1 : B { void f(); };  // ends the lifetime of *this
struct D2 : B { void f(); };  // undefined behavior

void B::mutate() {
    new (this) D2;            // reuses storage
    f();                      // undefined behavior
    ... = this;              // OK, this points to valid memory
}

void g() {
    void* p = std::malloc(sizeof(D1) + sizeof(D2));
    B* pb = new (p) D1;
    pb->mutate();
    *pb;                      // OK: pb points to valid memory
    void* q = pb;             // OK: pb points to valid memory
    pb->f();                  // undefined behavior: lifetime of *pb has ended
}

-- end example
```

Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see 11.10.5. Otherwise, such a glvalue refers to allocated storage (6.7.5.5.2).
storage (6.7.5.2), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

- the glvalue is used to access the object, or
- the glvalue is used to call a non-static member function of the object, or
- the glvalue is bound to a reference to a virtual base class (9.4.4), or
- the glvalue is used as the operand of a `dynamic_cast` (7.6.1.7) or as the operand of `typeid`.

If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if the original object is transparently replaceable (see below) by the new object. An object \(o_1\) is **transparently replaceable** by an object \(o_2\) if:

- the storage that \(o_2\) occupies exactly overlays the storage that \(o_1\) occupied, and
- \(o_1\) and \(o_2\) are of the same type (ignoring the top-level cv-qualifiers), and
- \(o_1\) is not a complete const object, and
- neither \(o_1\) nor \(o_2\) is a potentially-overlapping subobject (6.7.2), and
- either \(o_1\) and \(o_2\) are both complete objects, or \(o_1\) and \(o_2\) are direct subobjects of objects \(p_1\) and \(p_2\), respectively, and \(p_1\) is transparently replaceable by \(p_2\).

**Example 2:**
```c
struct C {
    int i;
    void f();
    const C& operator=( const C& );
};

const C& C::operator=( const C& other) {
if ( this != &other ) {
    this->~C(); // lifetime of *this ends
    new (this) C(other); // new object of type C created
    f(); // well-defined
} return *this;
}
C c1;
C c2;
c1 = c2; // well-defined
cl.f(); // well-defined; c1 refers to a new object of type C
```

**Note 3:** If these conditions are not met, a pointer to the new object can be obtained from a pointer that represents the address of its storage by calling `std::launder` (17.6.5). —end note

If a program ends the lifetime of an object of type \(T\) with static (6.7.5.2), thread (6.7.5.3), or automatic (6.7.5.4) storage duration and if \(T\) has a non-trivial destructor, \(^{34}\) and another object of the original type does not occupy that same storage location when the implicit destructor call takes place, the behavior of the program is undefined. This is true even if the block is exited with an exception.

**Example 3:**
```c
class T { }
struct B {
    ~B();
};
```

\(^{34}\) That is, an object for which a destructor will be called implicitly—upon exit from the block for an object with automatic storage duration, upon exit from the thread for an object with thread storage duration, or upon exit from the program for an object with static storage duration.
Creating a new object within the storage that a const complete object with static, thread, or automatic storage duration occupies, or within the storage that such a const object used to occupy before its lifetime ended, results in undefined behavior.

```
struct B {
    B();
    ~B();
};

const B b;

void h() {
    b.=B();
    new (const_cast<B*>(&b)) const B;  // undefined behavior
}
```

In this subclause, “before” and “after” refer to the “happens before” relation (6.9.2).

[Note 4: Therefore, undefined behavior results if an object that is being constructed in one thread is referenced from another thread without adequate synchronization. — end note]

### 6.7.4 Indeterminate values

[basic.indet]

1. When storage for an object with automatic or dynamic storage duration is obtained, the object has an indeterminate value, and if no initialization is performed for the object, that object retains an indeterminate value until that value is replaced (7.6.19).

   [Note 1: Objects with static or thread storage duration are zero-initialized, see 6.9.3.2. — end note]

2. If an indeterminate value is produced by an evaluation, the behavior is undefined except in the following cases:

   (2.1) — If an indeterminate value of unsigned ordinary character type (6.8.2) or std::byte type (17.2.1) is produced by the evaluation of:

   (2.1.1) — the second or third operand of a conditional expression (7.6.16),

   (2.1.2) — the right operand of a comma expression (7.6.20),

   (2.1.3) — the operand of a cast or conversion (7.3.9, 7.6.1.4, 7.6.1.9, 7.6.3) to an unsigned ordinary character type or std::byte type (17.2.1), or

   (2.1.4) — a discarded-value expression (7.2.3),

   then the result of the operation is an indeterminate value.

   (2.2) — If an indeterminate value of unsigned ordinary character type or std::byte type is produced by the evaluation of the right operand of a simple assignment operator (7.6.19) whose first operand is an lvalue of unsigned ordinary character type or std::byte type, an indeterminate value replaces the value of the object referred to by the left operand.

   (2.3) — If an indeterminate value of unsigned ordinary character type is produced by the evaluation of the initialization expression when initializing an object of unsigned ordinary character type, that object is initialized to an indeterminate value.

   (2.4) — If an indeterminate value of unsigned ordinary character type or std::byte type is produced by the evaluation of the initialization expression when initializing an object of std::byte type, that object is initialized to an indeterminate value.

[Example 1:]

```cpp
int f(bool b) {
    unsigned char c;
```
unsigned char d = c;  // OK, d has an indeterminate value
int e = d;  // undefined behavior
return b ? d : 0;  // undefined behavior if b is true

— end example —

6.7.5  Storage duration

6.7.5.1  General

The storage duration is the property of an object that defines the minimum potential lifetime of the storage containing the object. The storage duration is determined by the construct used to create the object and is one of the following:

1. static storage duration
2. thread storage duration
3. automatic storage duration
4. dynamic storage duration

Static, thread, and automatic storage durations are associated with objects introduced by declarations (6.2) and implicitly created by the implementation (6.7.7). The dynamic storage duration is associated with objects created by a new-expression (7.6.2.8).

The storage duration categories apply to references as well.

When the end of the duration of a region of storage is reached, the values of all pointers representing the address of any part of that region of storage become invalid pointer values (6.8.3). Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior. Any other use of an invalid pointer value has implementation-defined behavior.

6.7.5.2  Static storage duration

All variables which do not have dynamic storage duration, do not have thread storage duration, and are not local have static storage duration. The storage for these entities lasts for the duration of the program (6.9.3.2, 6.9.3.4).

If a variable with static storage duration has initialization or a destructor with side effects, it shall not be eliminated even if it appears to be unused, except that a class object or its copy/move may be eliminated as specified in 11.10.6.

The keyword static can be used to declare a local variable with static storage duration.

[Note 1: 8.8 describes the initialization of local static variables; 6.9.3.4 describes the destruction of local static variables. — end note]

The keyword static applied to a class data member in a class definition gives the data member static storage duration.

6.7.5.3  Thread storage duration

All variables declared with the thread_local keyword have thread storage duration. The storage for these entities lasts for the duration of the thread in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the entity associated with the current thread.

[Note 1: A variable with thread storage duration is initialized as specified in 6.9.3.2, 6.9.3.3, and 8.8 and, if constructed, is destroyed on thread exit (6.9.3.4). — end note]

6.7.5.4  Automatic storage duration

Block-scope variables not explicitly declared static, thread_local, or extern have automatic storage duration. The storage for these entities lasts until the block in which they are created exits.

[Note 1: These variables are initialized and destroyed as described in 8.8. — end note]

If a variable with automatic storage duration has initialization or a destructor with side effects, an implementation shall not destroy it before the end of its block nor eliminate it as an optimization, even if it appears to be unused, except that a class object or its copy/move may be eliminated as specified in 11.10.6.

Some implementations might define that copying an invalid pointer value causes a system-generated runtime fault.
6.7.5.5 Dynamic storage duration

6.7.5.5.1 General

Objects can be created dynamically during program execution (6.9.1), using `new-expression` (7.6.2.8), and destroyed using `delete-expression` (7.6.2.9). A C++ implementation provides access to, and management of, dynamic storage via the global allocation functions `operator new` and `operator new[]` and the global deallocation functions `operator delete` and `operator delete[]`.

[Note 1: The non-allocating forms described in 17.6.3.4 do not perform allocation or deallocation. —end note]

The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions are replaceable (17.6.3). A C++ program shall provide at most one definition of a replaceable allocation or deallocation function. Any such function definition replaces the default version provided in the library (16.4.5.6). The following allocation and deallocation functions (17.6) are implicitly declared in global scope in each translation unit of a program.

```cpp
[[nodiscard]] void* operator new(std::size_t);
[[nodiscard]] void* operator new(std::size_t, std::align_val_t);
void operator delete(void*) noexcept;
void operator delete(void*, std::size_t) noexcept;
void operator delete(void*, std::align_val_t) noexcept;
void operator delete(void*, std::size_t, std::align_val_t) noexcept;
[[nodiscard]] void* operator new[](std::size_t);
[[nodiscard]] void* operator new[](std::size_t, std::align_val_t);
void operator delete[](void*) noexcept;
void operator delete[](void*, std::size_t) noexcept;
void operator delete[](void*, std::align_val_t) noexcept;
void operator delete[](void*, std::size_t, std::align_val_t) noexcept;
```

These implicit declarations introduce only the function names `operator new`, `operator new[]`, `operator delete`, and `operator delete[]`.

[Note 2: The implicit declarations do not introduce the names `std`, `std::size_t`, `std::align_val_t`, or any other names that the library uses to declare these names. Thus, a `new-expression`, `delete-expression`, or function call that refers to one of these functions without importing or including the header `<new>` (17.6.2) is well-formed. However, referring to `std` or `std::size_t` or `std::align_val_t` is ill-formed unless the name has been declared by importing or including the appropriate header. —end note]

Allocation and/or deallocation functions may also be declared and defined for any class (11.12).

3 If the behavior of an allocation or deallocation function does not satisfy the semantic constraints specified in 6.7.5.5.2 and 6.7.5.5.3, the behavior is undefined.

6.7.5.5.2 Allocation functions

An allocation function shall be a class member function or a global function; a program is ill-formed if an allocation function is declared in a namespace scope other than global scope or declared static in global scope. The return type shall be `void*`. The first parameter shall have type `std::size_t` (17.2). The first parameter shall not have an associated default argument (9.3.4.7). The value of the first parameter is interpreted as the requested size of the allocation. An allocation function can be a function template. Such a template shall declare its return type and first parameter as specified above (that is, template parameter types shall not be used in the return type and first parameter type). Template allocation functions shall have two or more parameters.

An allocation function attempts to allocate the requested amount of storage. If it is successful, it returns the address of the start of a block of storage whose length in bytes is at least as large as the requested size. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned by a replaceable allocation function is a non-null pointer value (6.8.3) p0 different from any previously returned value p1, unless that value p1 was subsequently passed to a replaceable deallocation function. Furthermore, for the library allocation functions in 17.6.3.2 and 17.6.3.3, p0 represents the address
of a block of storage disjoint from the storage for any other object accessible to the caller. The effect of
indirecting through a pointer returned from a request for zero size is undefined.\textsuperscript{36}

For an allocation function other than a reserved placement allocation function (17.6.3.4), the pointer returned
on a successful call shall represent the address of storage that is aligned as follows:

\begin{enumerate}
  \item If the allocation function takes an argument of type `\texttt{std::align_val_t}`, the storage will have the
       alignment specified by the value of this argument.
  \item Otherwise, if the allocation function is named `\texttt{operator new[]}`, the storage is aligned for any object
       that does not have new-extended alignment (6.7.6) and is no larger than the requested size.
  \item Otherwise, the storage is aligned for any object that does not have new-extended alignment and is of
       the requested size.
\end{enumerate}

An allocation function that fails to allocate storage can invoke the currently installed new-handler function
(17.6.4.3), if any.

\textit{Note 1:} A program-supplied allocation function can obtain the address of the currently installed `\texttt{new_handler}`
using the `\texttt{std::get_new_handler}` function (17.6.4.5). \textit{— end note}\n
An allocation function that has a non-throwing exception specification (14.5) indicates failure by returning a
null pointer value. Any other allocation function never returns a null pointer value and indicates failure only by
throwing an exception (14.2) of a type that would match a handler (14.4) of type `\texttt{std::bad_alloc}` (17.6.4.1).

A global allocation function is only called as the result of a new expression (7.6.2.8), or called directly using
the function call syntax (7.6.1.3), or called indirectly to allocate storage for a coroutine state (9.5.4), or called
indirectly through calls to the functions in the C++ standard library.

\textit{Note 2:} In particular, a global allocation function is not called to allocate storage for objects with static storage
duration (6.7.5.2), for objects or references with thread storage duration (6.7.5.3), for objects of type `\texttt{std::type_info}`
(7.6.1.8), or for an exception object (14.2). \textit{— end note}\n
### 6.7.5.5.3 Deallocation functions

Deallocation functions shall be class member functions or global functions; a program is ill-formed if
deallocation functions are declared in a namespace scope other than global scope or declared static in global
scope.

A deallocation function is a \textit{destroying operator delete} if it has at least two parameters and its second
parameter is of type `\texttt{std::destroying_delete_t}`. A destroying operator delete shall be a class member
function named `\texttt{operator delete}`.

\textit{Note 1:} Array deletion cannot use a destroying operator delete. \textit{— end note}\n
Each deallocation function shall return `\texttt{void}`. If the function is a destroying operator delete declared in class
type `\texttt{C}` the type of its first parameter shall be `\texttt{C*}`; otherwise, the type of its first parameter shall be `\texttt{void*}`. A
deallocation function may have more than one parameter. A \textit{usual deallocation function} is a deallocation
function whose parameters after the first are

\begin{enumerate}
  \item optionally, a parameter of type `\texttt{std::destroying_delete_t}`, then
  \item optionally, a parameter of type `\texttt{std::size_t}`\textsuperscript{37}, then
  \item optionally, a parameter of type `\texttt{std::align_val_t}`.
\end{enumerate}

A destroying operator delete shall be a usual deallocation function. A deallocation function may be an
instance of a function template. Neither the first parameter nor the return type shall depend on a template
parameter. A deallocation function template shall have two or more function parameters. A template instance
is never a usual deallocation function, regardless of its signature.

If a deallocation function terminates by throwing an exception, the behavior is undefined. The value of the
first argument supplied to a deallocation function may be a null pointer value; if so, and if the deallocation
function is one supplied in the standard library, the call has no effect.

If the argument given to a deallocation function in the standard library is a pointer that is not the null
pointer value (6.8.3), the deallocation function shall deallocate the storage referenced by the pointer, ending
the duration of the region of storage.

\textsuperscript{36} The intent is to have `\texttt{operator new()}` implementable by calling `\texttt{std::malloc()}` or `\texttt{std::calloc()}`, so the rules are substan-
tially the same. C++ differs from C in requiring a zero request to return a non-null pointer.

\textsuperscript{37} The global `\texttt{operator delete(void*, std::size_t)}` precludes use of an allocation function `\texttt{void operator new(std::size_t, std::size_t)}` as a placement allocation function (C.3.3).

\textsection{6.7.5.5.3}
6.7.5.5.4 Safely-derived pointers

A traceable pointer object is

1. an object of an object pointer type (6.8.3), or
2. an object of an integral type that is at least as large as std::intptr_t, or
3. a sequence of elements in an array of narrow character type (6.8.2), where the size and alignment of the sequence match those of some object pointer type.

A pointer value is a safely-derived pointer to an object with dynamic storage duration only if the pointer value has an object pointer type and is one of the following:

1. the value returned by a call to the C++ standard library implementation of ::operator new(std::size_t) or ::operator new(std::size_t, std::align_val_t);
2. the result of taking the address of an object (or one of its subobjects) designated by an lvalue resulting from indirection through a safely-derived pointer value;
3. the result of well-defined pointer arithmetic (7.6.6) using a safely-derived pointer value;
4. the result of a well-defined pointer conversion (7.3.12, 7.6.1.4, 7.6.1.9, 7.6.3) of a safely-derived pointer value;
5. the result of a reinterpret_cast of a safely-derived pointer value;
6. the result of a reinterpret_cast of an integer representation of a safely-derived pointer value;
7. the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained a copy of a safely-derived pointer value.

3. An integer value is an integer representation of a safely-derived pointer only if its type is at least as large as std::intptr_t and it is one of the following:

1. the result of a reinterpret_cast of a safely-derived pointer value;
2. the result of a valid conversion of an integer representation of a safely-derived pointer value;
3. the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained an integer representation of a safely-derived pointer value;
4. the result of an additive or bitwise operation, one of whose operands is an integer representation of a safely-derived pointer value \(P\), if that result converted by reinterpret_cast<void*>(P) would compare equal to a safely-derived pointer computable from reinterpret_cast<void*>(P).

4. An implementation may have relaxed pointer safety, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value. Alternatively, an implementation may have strict pointer safety, in which case a pointer value referring to an object with dynamic storage duration that is not a safely-derived pointer value is an invalid pointer value unless the referenced complete object has previously been declared reachable (20.10.5).

[Note 1: The effect of using an invalid pointer value (including passing it to a deallocation function) is undefined, see 6.7.5. This is true even if the unsafely-derived pointer value might compare equal to some safely-derived pointer value. — end note]

It is implementation-defined whether an implementation has relaxed or strict pointer safety.

6.7.5.6 Duration of subobjects

The storage duration of subobjects and reference members is that of their complete object (6.7.2).

6.7.6 Alignment

Object types have alignment requirements (6.8.2, 6.8.3) which place restrictions on the addresses at which an object of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the alignment specifier (9.12.2).
A fundamental alignment is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to `alignof(std::max_align_t)` (17.2). The alignment required for a type may be different when it is used as the type of a complete object and when it is used as the type of a subobject.

*Example 1:*

```cpp
class B { long double d; };
class D : virtual B { char c; };
```

When `D` is the type of a complete object, it will have a subobject of type `B`, so it must be aligned appropriately for a `long double`. If `D` appears as a subobject of another object that also has `B` as a virtual base class, the `B` subobject might be part of a different subobject, reducing the alignment requirements on the `D` subobject. —end example

The result of the `alignof` operator reflects the alignment requirement of the type in the complete-object case.

An extended alignment is represented by an alignment greater than `alignof(std::max_align_t)`. It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported (9.12.2). A type having an extended alignment requirement is an over-aligned type.

[Note 1: Every over-aligned type is or contains a class type to which extended alignment applies (possibly through a non-static data member). — end note]

A new-extended alignment is represented by an alignment greater than `__STDCPP_DEFAULT_NEW_ALIGNMENT__` (15.11).

Alignments are represented as values of the type `std::size_t`. Valid alignments include only those values returned by an `alignof` expression for the fundamental types plus an additional implementation-defined set of values, which may be empty. Every alignment value shall be a non-negative integral power of two.

Alignments have an order from weaker to stronger or stricter alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

The alignment requirement of a complete type can be queried using an `alignof` expression (7.6.2.6). Furthermore, the narrow character types (6.8.2) shall have the weakest alignment requirement.

[Note 2: This enables the ordinary character types to be used as the underlying type for an aligned memory area (9.12.2). — end note]

Comparing alignments is meaningful and provides the obvious results:

- Two alignments are equal when their numeric values are equal.
- Two alignments are different when their numeric values are not equal.
- When an alignment is larger than another it represents a stricter alignment.

[Note 3: The runtime pointer alignment function (20.10.6) can be used to obtain an aligned pointer within a buffer; the aligned-storage templates in the library (20.15.8.7) can be used to obtain aligned storage. — end note]

If a request for a specific extended alignment in a specific context is not supported by an implementation, the program is ill-formed.

### 6.7.7 Temporary objects

Temporary objects are created

- when a prvalue is converted to an xvalue (7.3.5),
- when needed by the implementation to pass or return an object of trivially copyable type (see below), and
- when throwing an exception (14.2).

[Note 1: The lifetime of exception objects is described in 14.2. — end note]

Even when the creation of the temporary object is unevaluated (7.2), all the semantic restrictions shall be respected as if the temporary object had been created and later destroyed.

[Note 2: This includes accessibility (11.9) and whether it is deleted, for the constructor selected and for the destructor. However, in the special case of the operand of a `decltype-specifier` (9.2.9.5), no temporary is introduced, so the foregoing does not apply to such a prvalue. — end note]

The materialization of a temporary object is generally delayed as long as possible in order to avoid creating unnecessary temporary objects.

§ 6.7.7 69
Note 3: Temporary objects are materialized:

(2.1) — when binding a reference to a prvalue (9.4.4, 7.6.1.4, 7.6.1.7, 7.6.1.9, 7.6.1.11, 7.6.3),
(2.2) — when performing member access on a class prvalue (7.6.1.5, 7.6.4),
(2.3) — when performing an array-to-pointer conversion or subscripting on an array prvalue (7.3.3, 7.6.1.2),
(2.4) — when initializing an object of type `std::initializer_list<T>` from a braced-init-list (9.4.5),
(2.5) — for certain unevaluated operands (7.6.1.8, 7.6.2.5), and
(2.6) — when a prvalue that has type other than `cv void` appears as a discarded-value expression (7.2).

—end note

Example 1: Consider the following code:

```cpp
class X {
public:
    X(int);
    X(const X&);
    X& operator=(const X&);
    ~X();
};

class Y {
public:
    Y(int);
    Y(Y&&);
    ~Y();
};

X f(X);
Y g(Y);

void h() {
    X a(1);
    X b = f(X(2));
    Y c = g(Y(3));
    a = f(a);
}
```

`X(2)` is constructed in the space used to hold `f()`’s argument and `Y(3)` is constructed in the space used to hold `g()`’s argument. Likewise, `f()`’s result is constructed directly in `b` and `g()`’s result is constructed directly in `c`. On the other hand, the expression `a = f(a)` requires a temporary for the result of `f(a)`, which is materialized so that the reference parameter of `X::operator=(const X&)` can bind to it. —end example]

3 When an object of class type `X` is passed to or returned from a function, if `X` has at least one eligible copy or move constructor (11.4.4), each such constructor is trivial, and the destructor of `X` is either trivial or `deleted`, implementations are permitted to create a temporary object to hold the function parameter or result object. The temporary object is constructed from the function argument or return value, respectively, and the function’s parameter or return object is initialized as if by using the eligible trivial constructor to copy the temporary (even if that constructor is inaccessible or would not be selected by overload resolution to perform a copy or move of the object).

[Note 4: This latitude is granted to allow objects of class type to be passed to or returned from functions in registers. —end note]

4 When an implementation introduces a temporary object of a class that has a non-trivial constructor (11.4.5.2, 11.4.5.3), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (11.4.7). Temporary objects are destroyed as the last step in evaluating the full-expression (6.9.1) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception. The value computations and side effects of destroying a temporary object are associated only with the full-expression, not with any specific subexpression.

5 There are three contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array with no corresponding initializer (9.4). The second context is when a copy constructor is called to copy an element of an array while the entire array is copied (7.5.5.3, 11.4.5.3). In either case, if the constructor has one or
The destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.

The third context is when a reference is bound to a temporary object. The temporary object to which the reference is bound or the temporary object that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference if the glvalue to which the reference is bound was obtained through one of the following:

- a temporary materialization conversion (7.3.5),
- (expression), where expression is one of these expressions,
- subscripting (7.6.1.2) of an array operand, where that operand is one of these expressions,
- a class member access (7.6.1.5) using the . operator where the left operand is one of these expressions and the right operand designates a non-static data member of non-reference type,
- a pointer-to-member operation (7.6.4) using the .* operator where the left operand is one of these expressions and the right operand is a pointer to data member of non-reference type,
- a
  - const_cast (7.6.1.11),
  - static_cast (7.6.1.9),
  - dynamic_cast (7.6.1.7), or
  - reinterpret_cast (7.6.1.10)
    converting, without a user-defined conversion, a glvalue operand that is one of these expressions to a glvalue that refers to the object designated by the operand, or to its complete object or a subobject thereof,
- a conditional expression (7.6.16) that is a glvalue where the second or third operand is one of these expressions, or
- a comma expression (7.6.20) that is a glvalue where the right operand is one of these expressions.

[Example 2:]
```
template<typename T> using id = T;

int i = 1;
int&& a = id<int[3]>{1, 2, 3}[i]; // temporary array has same lifetime as a
const int& b = static_cast<const int&>(0); // temporary int has same lifetime as b
int&& c = cond ? id<int[3]>{1, 2, 3}[i] : static_cast<int&&>(0); // exactly one of the two temporaries is lifetime-extended
```

[Note 5: An explicit type conversion (7.6.1.4, 7.6.3) is interpreted as a sequence of elementary casts, covered above.

[Example 3:]
```
const int& x = (const int&)1; // temporary for value 1 has same lifetime as x
```

[Note 6: If a temporary object has a reference member initialized by another temporary object, lifetime extension applies recursively to such a member’s initializer.

[Example 4:]
```
struct S {
    const int& m;
};
const S& s = S{1}; // both S and int temporaries have lifetime of s
```

The exceptions to this lifetime rule are:

---

39 The same rules apply to initialization of an initializer_list object (9.4.5) with its underlying temporary array.
- A temporary object bound to a reference parameter in a function call (7.6.1.3) persists until the completion of the full-expression containing the call.

- A temporary object bound to a reference element of an aggregate of class type initialized from a parenthesized expression-list (9.4) persists until the completion of the full-expression containing the expression-list.

- The lifetime of a temporary bound to the returned value in a function return statement (8.7.4) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.

- A temporary bound to a reference in a new-initializer (7.6.2.8) persists until the completion of the full-expression containing the new-initializer.

[Note 7: This might introduce a dangling reference. — end note]

Example 5:

```cpp
struct S { int mi; const std::pair<int,int>& mp; }
S a { 1, {2,3} }
S* p = new S{ 1, {2,3} }; // creates dangling reference
```

The destruction of a temporary whose lifetime is not extended by being bound to a reference is sequenced before the destruction of every temporary which is constructed earlier in the same full-expression. If the lifetime of two or more temporaries to which references are bound ends at the same point, these temporaries are destroyed at that point in the reverse order of the completion of their construction. In addition, the destruction of temporaries bound to references shall take into account the ordering of destruction of objects with static, thread, or automatic storage duration (6.7.5.2, 6.7.5.3, 6.7.5.4); that is, if `obj1` is an object with the same storage duration as the temporary and created before the temporary is created the temporary shall be destroyed before `obj1` is destroyed; if `obj2` is an object with the same storage duration as the temporary and created after the temporary is created the temporary shall be destroyed after `obj2` is destroyed.

Example 6:

```cpp
struct S {
    S();
    S(int);
    friend S operator+(const S&, const S&);
    ~S();
};
S obj1;
const S& cr = S(16)+S(23);
S obj2;
```

The expression `S(16) + S(23)` creates three temporaries: a first temporary `T1` to hold the result of the expression `S(16)`, a second temporary `T2` to hold the result of the expression `S(23)`, and a third temporary `T3` to hold the result of the addition of these two expressions. The temporary `T3` is then bound to the reference `cr`. It is unspecified whether `T1` or `T2` is created first. On an implementation where `T1` is created before `T2`, `T2` shall be destroyed before `T1`. The temporaries `T1` and `T2` are bound to the reference parameters of `operator+`; these temporaries are destroyed at the end of the full-expression containing the call to `operator+`. The temporary `T3` bound to the reference `cr` is destroyed at the end of `cr`'s lifetime, that is, at the end of the program. In addition, the order in which `T3` is destroyed takes into account the destruction order of other objects with static storage duration. That is, because `obj1` is constructed before `T3`, and `T3` is constructed before `obj2`, `obj2` shall be destroyed before `T3`, and `T3` shall be destroyed before `obj1`. — end example

6.8 Types

6.8.1 General

[Note 1: 6.8 and the subclauses thereof impose requirements on implementations regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects (6.7.2), references (9.3.4.3), or functions (9.3.4.6). — end note]

For any object (other than a potentially-overlapping subobject) of trivially copyable type `T`, whether or not the object holds a valid value of type `T`, the underlying bytes (6.7.1) making up the object can be copied into an array of `char`, `unsigned char`, or `std::byte` (17.2.1). If the content of that array is copied back into the object, the object shall subsequently hold its original value.

40) By using, for example, the library functions (16.4.2.3) `std::memcpys` or `std::memmove`.

§ 6.8.1
Example 1:

```cpp
constexpr std::size_t N = sizeof(T);
char buf[N];  // obj initialized to its original value
T obj;       // between these two calls to std::memcpy, obj might be modified
std::memcpy(buf, &obj, N);  // at this point, each subobject of obj of scalar type holds its original value
std::memcpy(&obj, buf, N);
```

—end example

For any trivially copyable type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, where neither `obj1` nor `obj2` is a potentially-overlapping subobject, if the underlying bytes (6.7.1) making up `obj1` are copied into `obj2`, \(^{41}\) `obj2` shall subsequently hold the same value as `obj1`.

Example 2:

```cpp
T* t1p;
T* t2p;
// provided that t2p points to an initialized object ...
std::memcpy(t1p, t2p, sizeof(T));  // at this point, every subobject of trivially copyable type in *t1p contains
// the same value as the corresponding subobject in *t2p
```

—end example

The object representation of an object of type `T` is the sequence of \(N\) unsigned `char` objects taken up by the object of type `T`, where \(N\) equals `sizeof(T)`. The value representation of an object of type `T` is the set of bits that participate in representing a value of type `T`. Bits in the object representation that are not part of the value representation are padding bits. For trivially copyable types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.\(^{42}\)

A class that has been declared but not defined, an enumeration type in certain contexts (9.7.1), or an array of unknown bound or of incomplete element type, is an incompletely-defined object type.\(^{43}\) Incompletely-defined object types and cv `void` are incomplete types (6.8.2).

[Note 2: Objects cannot be defined to have an incomplete type (6.2). —end note]

A class type (such as \"class X\") can be incomplete at one point in a translation unit and complete later on; the type \"class X\" is the same type at both points. The declared type of an array object can be an array of incomplete class type and therefore incomplete; if the class type is completed later on in the translation unit, the array type becomes complete; the array type at those two points is the same type. The declared type of an array object can be an array of unknown bound and therefore be incomplete at one point in a translation unit and complete later on; the array types at those two points \("array of unknown bound of `T`" and \"array of `N` `T`\") are different types. The type of a pointer to array of unknown bound, or of a type defined by a `typedef` declaration to be an array of unknown bound, cannot be completed.

Example 3:

```cpp
class X;  // X is an incomplete type
extern X* xp;  // xp is a pointer to an incomplete type
extern int arr[];  // the type of arr is incomplete
typedef int UNKA[];  // UNKA is an incomplete type
UNKA* arrp;  // arrp is a pointer to an incomplete type
UNKA** arrpp;

void foo() {
    xp++;  // error: X is incomplete
    arrp++;  // error: incomplete type
    arrpp++;  // OK: sizeof UNKA* is known
}

struct X { int i; };  // now X is a complete type
int arr[10];  // now the type of arr is complete
```

\(^{41}\) By using, for example, the library functions (16.4.2.3) `std::memcpy` or `std::memmove`.

\(^{42}\) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.

\(^{43}\) The size and layout of an instance of an incompletely-defined object type is unknown.
X x;
void bar() {
    xp = &x;  // OK; type is “pointer to X”
    arrp = &arr;  // error: different types
    xp++;
    // OK: X is complete
    arrp++;
    // error: UNKA can’t be completed
}
—end example

[Note 3: The rules for declarations and expressions describe in which contexts incomplete types are prohibited. — end note]

An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not cv void.

Arithmetic types (6.8.2), enumeration types, pointer types, pointer-to-member types (6.8.3), std::nullptr_t, and cv-qualified (6.8.4) versions of these types are collectively called scalar types. Scalar types, trivially copyable class types (11.2), arrays of such types, and cv-qualified versions of these types are collectively called trivially copyable types. Scalar types, trivial class types (11.2), arrays of such types and cv-qualified versions of these types are collectively called trivial types. Scalar types, standard-layout class types (11.2), arrays of such types and cv-qualified versions of these types are collectively called standard-layout types. Scalar types, implicit-lifetime class types (11.2), array types, and cv-qualified versions of these types are collectively called implicit-lifetime types.

A type is a literal type if it is:

(10.1) — cv void; or
(10.2) — a scalar type; or
(10.3) — a reference type; or
(10.4) — an array of literal type; or
(10.5) — a possibly cv-qualified class type (Clause 11) that has all of the following properties:
(10.5.1) — it has a constexpr destructor (9.2.6),
(10.5.2) — it is either a closure type (7.5.5.2), an aggregate type (9.4.2), or has at least one constexpr constructor or constructor template (possibly inherited (9.9) from a base class) that is not a copy or move constructor,
(10.5.3) — if it is a union, at least one of its non-static data members is of non-volatile literal type, and
(10.5.4) — if it is not a union, all of its non-static data members and base classes are of non-volatile literal types.

[Note 4: A literal type is one for which it might be possible to create an object within a constant expression. It is not a guarantee that it is possible to create such an object, nor is it a guarantee that any object of that type will be usable in a constant expression. — end note]

Two types cv1 T1 and cv2 T2 are layout-compatible types if T1 and T2 are the same type, layout-compatible enumerations (9.7.1), or layout-compatible standard-layout class types (11.4).

6.8.2 Fundamental types

There are five standard signed integer types: “signed char”, “short int”, “int”, “long int”, and “long long int”. In this list, each type provides at least as much storage as those preceding it in the list. There may also be implementation-defined extended signed integer types. The standard and extended signed integer types are collectively called signed integer types. The range of representable values for a signed integer type is $-2^{N-1}$ to $2^{N-1} - 1$ (inclusive), where $N$ is called the width of the type.

[Note 1: Plain ints are intended to have the natural width suggested by the architecture of the execution environment; the other signed integer types are provided to meet special needs. — end note]

For each of the standard signed integer types, there exists a corresponding (but different) standard unsigned integer type: “unsigned char”, “unsigned short int”, “unsigned int”, “unsigned long int”, and “unsigned long long int”. Likewise, for each of the extended signed integer types, there exists a corresponding extended unsigned integer type. The standard and extended unsigned integer types are collectively called unsigned integer types. An unsigned integer type has the same width $N$ as the corresponding signed
integer type. The range of representable values for the unsigned type is 0 to $2^N - 1$ (inclusive); arithmetic for the unsigned type is performed modulo $2^N$.

[Note 2: Unsigned arithmetic does not overflow. Overflow for signed arithmetic yields undefined behavior (7.1). —end note]

An unsigned integer type has the same object representation, value representation, and alignment requirements (6.7.6) as the corresponding signed integer type. For each value $x$ of a signed integer type, the value of the corresponding unsigned integer type congruent to $x$ modulo $2^N$ has the same value of corresponding bits in its value representation.\footnote{This is also known as two’s complement representation.}

[Example 1: The value $-1$ of a signed integer type has the same representation as the largest value of the corresponding unsigned type. —end example]

<table>
<thead>
<tr>
<th>Table 12: Minimum width</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>signed char</td>
</tr>
<tr>
<td>short int</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>long int</td>
</tr>
<tr>
<td>long long int</td>
</tr>
</tbody>
</table>

The width of each signed integer type shall not be less than the values specified in Table 12. The value representation of a signed or unsigned integer type comprises $N$ bits, where $N$ is the respective width. Each set of values for any padding bits (6.8) in the object representation are alternative representations of the value specified by the value representation.

[Note 3: Padding bits have unspecified value, but cannot cause traps. In contrast, see ISO C 6.2.6.2. —end note]
[Note 4: The signed and unsigned integer types satisfy the constraints given in ISO C 5.2.4.2.1. —end note]

Except as specified above, the width of a signed or unsigned integer type is implementation-defined.

Each value $x$ of an unsigned integer type with width $N$ has a unique representation $x = x_02^0 + x_12^1 + \ldots + x_{N-1}2^{N-1}$, where each coefficient $x_i$ is either 0 or 1; this is called the base-2 representation of $x$. The base-2 representation of a value of signed integer type is the base-2 representation of the congruent value of the corresponding unsigned integer type. The standard signed integer types and standard unsigned integer types are collectively called the standard integer types, and the extended signed integer types and extended unsigned integer types are collectively called the extended integer types.

A fundamental type specified to have a signed or unsigned integer type as its underlying type has the same object representation, value representation, alignment requirements (6.7.6), and range of representable values as the underlying type. Further, each value has the same representation in both types.

Type char is a distinct type that has an implementation-defined choice of “signed char” or “unsigned char” as its underlying type. The values of type char can represent distinct codes for all members of the implementation’s basic character set. The three types char, signed char, and unsigned char are collectively called ordinary character types. The ordinary character types and char8_t are collectively called narrow character types. For narrow character types, each possible bit pattern of the object representation represents a distinct value.

[Note 5: This requirement does not hold for other types. —end note]
[Note 6: A bit-field of narrow character type whose width is larger than the width of that type has padding bits; see 6.8. —end note]

Type wchar_t is a distinct type that has an implementation-defined signed or unsigned integer type as its underlying type. The values of type wchar_t can represent distinct codes for all members of the largest extended character set specified among the supported locales (28.3.1).

Type char8_t denotes a distinct type whose underlying type is unsigned char. Types char16_t and char32_t denote distinct types whose underlying types are uint_least16_t and uint_least32_t, respectively, in <cstdint> (17.4.2).
Type `bool` is a distinct type that has the same object representation, value representation, and alignment requirements as an implementation-defined unsigned integer type. The values of type `bool` are `true` and `false`.

[Note 7: There are no `signed`, `unsigned`, `short`, or `long` `bool` types or values. — end note]

Types `bool`, `char`, `wchar_t`, `char8_t`, `char16_t`, `char32_t`, and the signed and unsigned integer types are collectively called *integral types*. A synonym for integral type is *integer type*.

[Note 8: Enumerations (9.7.1) are not integral; however, unscoped enumerations can be promoted to integral types as specified in 7.3.7. — end note]

There are three *floating-point types*: `float`, `double`, and `long double`. The type `double` provides at least as much precision as `float`, and the type `long double` provides at least as much precision as `double`. The set of values of the type `float` is a subset of the set of values of the type `double`; the set of values of the type `double` is a subset of the set of values of the type `long double`. The value representation of floating-point types is implementation-defined.

[Note 9: This document imposes no requirements on the accuracy of floating-point operations; see also 17.3. — end note]

Integral and floating-point types are collectively called *arithmetic* types. Specializations of the standard library template `std::numeric_limits` (17.3) shall specify the maximum and minimum values of each arithmetic type for an implementation.

A type `cv void` is an incomplete type that cannot be completed; such a type has an empty set of values. It is used as the return type for functions that do not return a value. Any expression can be explicitly converted to type `cv void` (7.6.1.4, 7.6.1.9, 7.6.3). An expression of type `cv void` shall be used only as an expression statement (8.3), as an operand of a comma expression (7.6.20), as the second or third operand of `?:` (7.6.16), as the operand of `typeid`, `noexcept`, or `decltype`, as the expression in a `return` statement (8.7.4) for a function with the return type `cv void`, or as the operand of an explicit conversion to type `cv void`.

A value of type `std::nullptr_t` is a null pointer constant (7.3.12). Such values participate in the pointer and the pointer-to-member conversions (7.3.12, 7.3.13). `sizeof(std::nullptr_t)` shall be equal to `sizeof(void*)`.

The types described in this subclause are called *fundamental types*.

[Note 10: Even if the implementation defines two or more fundamental types to have the same value representation, they are nevertheless different types. — end note]

### 6.8.3 Compound types

[basic.compound]

Compound types can be constructed in the following ways:

1. **Arrays** of objects of a given type, 9.3.4.5;
2. **Functions**, which have parameters of given types and return `void` or references or objects of a given type, 9.3.4.6;
3. **Pointers** to `cv void` or objects or functions (including static members of classes) of a given type, 9.3.4.2;
4. **References** to objects or functions of a given type, 9.3.4.3. There are two types of references:
   1.4.1 **lvalue reference**
   1.4.2 **rvalue reference**
5. **Classes** containing a sequence of objects of various types (Clause 11), a set of types, enumerations and functions for manipulating these objects (11.4.2), and a set of restrictions on the access to these entities (11.9);
6. **Unions**, which are classes capable of containing objects of different types at different times, 11.5;
7. **Enumerations**, which comprise a set of named constant values. Each distinct enumeration constitutes a different enumerated type, 9.7.1;
8. **Pointers to non-static class members**, which identify members of a given type within objects of a given class, 9.3.4.4. Pointers to data members and pointers to member functions are collectively called *pointer-to-member* types.

---

45) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.
These methods of constructing types can be applied recursively; restrictions are mentioned in 9.3.4. Constructing a type such that the number of bytes in its object representation exceeds the maximum value representable in the type `std::size_t` (17.2) is ill-formed.

The type of a pointer to `cv void` or a pointer to an object type is called an object pointer type.

[Note 1: A pointer to `void` does not have a pointer-to-object type, however, because `void` is not an object type. — end note]

The type of a pointer that can designate a function is called a function pointer type. A pointer to an object of type `T` is referred to as a “pointer to `T`”.

[Example 1: A pointer to an object of type `int` is referred to as “pointer to `int`” and a pointer to an object of class `X` is called a “pointer to `X`”. — end example]

Except for pointers to static members, text referring to “pointers” does not apply to pointers to members. Pointers to incomplete types are allowed although there are restrictions on what can be done with them (6.7.6). Every value of pointer type is one of the following:

1. A pointer to an object or function (the pointer is said to point to the object or function), or
2. A pointer past the end of an object (7.6.6), or
3. The null pointer value for that type, or
4. An invalid pointer value.

A value of a pointer type that is a pointer to or past the end of an object represents the address of the first byte in memory (6.7.1) occupied by the object or the first byte in memory after the end of the storage occupied by the object, respectively.

[Note 2: A pointer past the end of an object (7.6.6) is not considered to point to an unrelated object of the object’s type that might be located at that address. A pointer value becomes invalid when the storage it denotes reaches the end of its storage duration; see 6.7.5. — end note]

For purposes of pointer arithmetic (7.6.6) and comparison (7.6.9, 7.6.10), a pointer past the end of the last element of an array `x` of `n` elements is considered to be equivalent to a pointer to a hypothetical array element `n` of `x` and an object of type `T` that is not an array element is considered to belong to an array with one element of type `T`. The value representation of pointer types is implementation-defined. Pointers to layout-compatible types shall have the same value representation and alignment requirements (6.7.6).

[Note 3: Pointers to over-aligned types (6.7.6) have no special representation, but their range of valid values is restricted by the extended alignment requirement. — end note]

Two objects `a` and `b` are pointer-interconvertible if:

1. They are the same object, or
2. One is a union object and the other is a non-static data member of that object (11.5), or
3. One is a standard-layout class object and the other is the first non-static data member of that object, or, if the object has no non-static data members, any base class subobject of that object (11.4), or
4. There exists an object `c` such that `a` and `c` are pointer-interconvertible, and `c` and `b` are pointer-interconvertible.

If two objects are pointer-interconvertible, then they have the same address, and it is possible to obtain a pointer to one from a pointer to the other via a reinterpret_cast (7.6.1.10).

[Note 4: An array object and its first element are not pointer-interconvertible, even though they have the same address. — end note]

A pointer to `cv void` can be used to point to objects of unknown type. Such a pointer shall be able to hold any object pointer. An object of type `cv void` shall have the same representation and alignment requirements as `cv char`.

### 6.8.4 CV-qualifiers

A type mentioned in 6.8.2 and 6.8.3 is a cv-unqualified type. Each type which is a cv-unqualified object type or is `void` (6.8) has three corresponding cv-qualified versions of its type: a `const-qualified` version, a `volatile-qualified` version, and a `const-volatile-qualified` version. The type of an object (6.7.2) includes the

46) For an object that is not within its lifetime, this is the first byte in memory that it will occupy or used to occupy.
cv-qualifiers specified in the decl-specifier-seq (9.2), declarator (9.3), type-id (9.3.2), or new-type-id (7.6.2.8) when the object is created.

1.1 — A const object is an object of type const T or a non-mutable subobject of a const object.
1.2 — A volatile object is an object of type volatile T or a subobject of a volatile object.
1.3 — A const volatile object is an object of type const volatile T, a non-mutable subobject of a const volatile object, a const subobject of a volatile object, or a non-mutable volatile subobject of a const object.

The cv-qualified or cv-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (6.7.6).

2 Except for array types, a compound type (6.8.3) is not cv-qualified by the cv-qualifiers (if any) of the types from which it is compounded.

3 An array type whose elements are cv-qualified is also considered to have the same cv-qualifications as its elements.

[Note 1: Cv-qualifiers applied to an array type attach to the underlying element type, so the notation “cv T”, where T is an array type, refers to an array whose elements are so-qualified (9.3.4.5). — end note]

[Example 1:

typedef char CA[5];
typedef const char CC;
CC arr1[5] = { 0 };
const CA arr2[5] = { 0 };

The type of both arr1 and arr2 is “array of 5 const char”, and the array type is considered to be const-qualified. — end example]

4 [Note 2: See 9.3.4.6 and 11.4.3.2 regarding function types that have cv-qualifiers. — end note]

5 There is a partial ordering on cv-qualifiers, so that a type can be said to be more cv-qualified than another. Table 13 shows the relations that constitute this ordering.

Table 13: Relations on const and volatile

| no cv-qualifier | < | const |
| no cv-qualifier | < | volatile |
| no cv-qualifier | < | const volatile |
| const | < | const volatile |
| volatile | < | const volatile |

6 In this document, the notation cv (or cv1, cv2, etc.), used in the description of types, represents an arbitrary set of cv-qualifiers, i.e., one of {const}, {volatile}, {const, volatile}, or the empty set. For a type cv T, the top-level cv-qualifiers of that type are those denoted by cv.

[Example 2: The type corresponding to the type-id const int* has no top-level cv-qualifiers. The type corresponding to the type-id volatile int * const has the top-level cv-qualifier const. For a class type C, the type corresponding to the type-id void (C::* volatile)(int) const has the top-level cv-qualifier volatile. — end example]

6.8.5 Integer conversion rank [conv.rank]

1 Every integer type has an integer conversion rank defined as follows:

1.1 — No two signed integer types other than char and signed char (if char is signed) shall have the same rank, even if they have the same representation.
1.2 — The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller width.
1.3 — The rank of long long int shall be greater than the rank of long int, which shall be greater than the rank of int, which shall be greater than the rank of short int, which shall be greater than the rank of signed char.
1.4 — The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.

47) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and non-static data members of unions.
The rank of any standard integer type shall be greater than the rank of any extended integer type with
the same width.

The rank of char shall equal the rank of signed char and unsigned char.

The rank of bool shall be less than the rank of all other standard integer types.

The ranks of char8_t, char16_t, char32_t, and wchar_t shall equal the ranks of their underlying
types (6.8.2).

The rank of any extended signed integer type relative to another extended signed integer type with the
same width is implementation-defined, but still subject to the other rules for determining the integer
conversion rank.

For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3,
then T1 shall have greater rank than T3.

[Note 1: The integer conversion rank is used in the definition of the integral promotions (7.3.7) and the usual
arithmetic conversions (7.4). — end note]

### 6.9 Program execution

#### 6.9.1 Sequential execution

An instance of each object with automatic storage duration (6.7.5.4) is associated with each entry into its
block. Such an object exists and retains its last-stored value during the execution of the block and while the
block is suspended (by a call of a function, suspension of a coroutine (7.6.2.4), or receipt of a signal).

A constituent expression is defined as follows:

1. The constituent expression of an expression is that expression.
2. The constituent expressions of a braced-init-list or of a (possibly parenthesized) expression-list are the
   constituent expressions of the elements of the respective list.
3. The constituent expressions of a brace-or-equal-initializer of the form = initializer-clause are the constituent
   expressions of the initializer-clause.

[Example 1:

```c
struct A { int x; };
struct B { int y; struct A a; };
B b = { 5, { 1+1 } };
```

The constituent expressions of the initializer used for the initialization of b are 5 and 1+1. — end example]

The immediate subexpressions of an expression E are

1. the constituent expressions of E’s operands (7.2),
2. any function call that E implicitly invokes,
3. if E is a lambda-expression (7.5.5), the initialization of the entities captured by copy and the constituent
   expressions of the elements of the respective list.
4. if E is a function call (7.6.1.3) or implicitly invokes a function, the constituent expressions of each
default argument (9.3.4.7) used in the call, or
5. if E creates an aggregate object (9.4.2), the constituent expressions of each default member initializer
   (11.4) used in the initialization.

A subexpression of an expression E is an immediate subexpression of E or a subexpression of an immediate
subexpression of E.

[Note 1: Expressions appearing in the compound-statement of a lambda-expression are not subexpressions of the
lambda-expression. — end note]

A full-expression is

1. an unevaluated operand (7.2),
2. a constant-expression (7.7),
3. an immediate invocation (7.7),
4. an init-declarator (9.3) or a mem-initializer (11.10.3), including the constituent expressions of the initializer,
— an invocation of a destructor generated at the end of the lifetime of an object other than a temporary object (6.7.7) whose lifetime has not been extended, or

— an expression that is not a subexpression of another expression and that is not otherwise part of a full-expression.

If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression. For an initializer, performing the initialization of the entity (including evaluating default member initializers of an aggregate) is also considered part of the full-expression.

[Example 2:

```cpp
struct S {
    S(int i): I(i) { }  // full-expression is initialization of I
    int& v() { return I; }
    ~S() noexcept(false) { }
    private:
        int I;
};

S s1(1);  // full-expression comprises call of S::S(int)
void f() {
    S s2 = 2;  // full-expression comprises call of S::S(int)
    if (S(3).v()) // full-expression includes lvalue-to-rvalue and int to bool conversions,
        // performed before temporary is deleted at end of full-expression
        {}
    bool b = noexcept(S());  // exception specification of destructor of S considered for noexcept
                         // full-expression is destruction of s2 at end of block
}
struct B {
    B(S = S(0));
};
B b[2] = { B(), B() };  // full-expression is the entire initialization
                         // including the destruction of temporaries

end example]

[Note 2: The evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. For example, subexpressions involved in evaluating default arguments (9.3.4.7) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — end note]

Reading an object designated by a `volatile` glvalue (7.2.1), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. Evaluation of an expression (or a subexpression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access through a volatile glvalue is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the `volatile` access may not have completed yet.

`Sequenced before` is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (6.9.2), which induces a partial order among those evaluations. Given any two evaluations `A` and `B`, if `A` is sequenced before `B` (or, equivalently, `B` is sequenced after `A`), then the execution of `A` shall precede the execution of `B`. If `A` is not sequenced before `B` and `B` is not sequenced before `A`, then `A` and `B` are unsequenced.

[Note 3: The execution of unsequenced evaluations can overlap. — end note]

Evaluations `A` and `B` are `indeterminately sequenced` when either `A` is sequenced before `B` or `B` is sequenced before `A`, but it is unspecified which.

[Note 4: Indeterminately sequenced evaluations cannot overlap, but either could be executed first. — end note]

An expression `X` is said to be sequenced before an expression `Y` if every value computation and every side effect associated with the expression `X` is sequenced before every value computation and every side effect associated with the expression `Y`.
Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.\footnote{As specified in 6.7.7, after a full-expression is evaluated, a sequence of zero or more invocations of destructor functions for temporary objects takes place, usually in reverse order of the construction of each temporary object.}

Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced.

[Note 5: In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. — end note]

The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a memory location (6.7.1) is unsequenced relative to either another side effect on the same memory location or a value computation using the value of any object in the same memory location, and they are not potentially concurrent (6.9.2), the behavior is undefined.

[Note 6: The next subclause imposes similar, but more complex restrictions on potentially concurrent computations. — end note]

Example 3:

```cpp
void g(int i) {
    i = 7, i++, i++;
    // i becomes 9
    i = i++ + 1;
    // the value of i is incremented
    i = i++ + i;
    // undefined behavior
    i = i + 1;
    // the value of i is incremented
}
```

When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. For each function invocation $F$, for every evaluation $A$ that occurs within $F$ and every evaluation $B$ that does not occur within $F$ but is evaluated on the same thread and as part of the same signal handler (if any), either $A$ is sequenced before $B$ or $B$ is sequenced before $A$.$^{49}$

[Note 7: If $A$ and $B$ would not otherwise be sequenced then they are indeterminately sequenced. — end note]

Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit.

Example 4: Evaluation of a new-expression invokes one or more allocation and constructor functions; see 7.6.2.8. For another example, invocation of a conversion function (11.4.8.3) can arise in contexts in which no function call syntax appears. — end example]

The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, regardless of the syntax of the expression that calls the function.

If a signal handler is executed as a result of a call to the `std::raise` function, then the execution of the handler is sequenced after the invocation of the `std::raise` function and before its return.

[Note 8: When a signal is received for another reason, the execution of the signal handler is usually unsequenced with respect to the rest of the program. — end note]

6.9.2 Multi-threaded executions and data races

6.9.2.1 General

A thread of execution (also known as a thread) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread.

[Note 1: When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. — end note]

Every thread in a program can potentially access every object and function in a program.$^{50}$ Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each

\footnote{In other words, function executions do not interleave with each other.}

\footnote{An object with automatic or thread storage duration (6.7.5) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (6.8.3).}
thread proceeds as defined by the remainder of this document. The execution of the entire program consists of an execution of all of its threads.

[Note 2: Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — end note]

Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

2 For a signal handler that is not executed as a result of a call to the std::raise function, it is unspecified which thread of execution contains the signal handler invocation.

6.9.2.2 Data races

1 The value of an object visible to a thread \( T \) at a particular point is the initial value of the object, a value assigned to the object by \( T \), or a value assigned to the object by another thread, according to the rules below.

[Note 1: In some cases, there might instead be undefined behavior. Much of this subclause is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — end note]

2 Two expression evaluations conflict if one of them modifies a memory location (6.7.1) and the other one reads or modifies the same memory location.

3 The library defines a number of atomic operations (Clause 31) and operations on mutexes (Clause 32) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics.

[Note 2: For example, a call that acquires a mutex will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on \( A \) forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on \( A \). “Relaxed” atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. — end note]

4 All modifications to a particular atomic object \( M \) occur in some particular total order, called the modification order of \( M \).

[Note 3: There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads might observe modifications to different objects in inconsistent orders. — end note]

5 A release sequence headed by a release operation \( A \) on an atomic object \( M \) is a maximal contiguous subsequence of side effects in the modification order of \( M \), where the first operation is \( A \), and every subsequent operation is an atomic read-modify-write operation.

6 Certain library calls synchronize with other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store (31.4).

[Note 4: Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. — end note]

[Note 5: The specifications of the synchronization operations define when one reads the value written by another. For atomic objects, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition “reads the value written” by the last mutex release. — end note]

7 An evaluation \( A \) carries a dependency to an evaluation \( B \) if

(7.1) — the value of \( A \) is used as an operand of \( B \), unless:

(7.1.1) — \( B \) is an invocation of any specialization of std::kill_dependency (31.4), or

(7.1.2) — \( A \) is the left operand of a built-in logical AND (&&, see 7.6.14) or logical OR (||, see 7.6.15) operator, or

(7.1.3) — \( A \) is the left operand of a conditional (?:, see 7.6.16) operator, or

(7.1.4) — \( A \) is the left operand of the built-in comma (,) operator (7.6.20);
or

(7.2)  — A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or

(7.3)  — for some evaluation X, A carries a dependency to X, and X carries a dependency to B.

[Note 6: “Carries a dependency to” is a subset of “is sequenced before”, and is similarly strictly intra-thread. — end note]

8 An evaluation A is dependency-ordered before an evaluation B if

(8.1)  — A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads the value written by A, or

(8.2)  — for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

[Note 7: The relation “is dependency-ordered before” is analogous to “synchronizes with”, but uses release/consume in place of release/acquire. — end note]

9 An evaluation A inter-thread happens before an evaluation B if

(9.1)  — A synchronizes with B, or

(9.2)  — A is dependency-ordered before B, or

(9.3)  — for some evaluation X

(9.3.1) — A synchronizes with X and X is sequenced before B, or

(9.3.2) — A is sequenced before X and X inter-thread happens before B, or

(9.3.3) — A inter-thread happens before X and X inter-thread happens before B.

[Note 8: The “inter-thread happens before” relation describes arbitrary concatenations of “sequenced before”, “synchronizes with” and “dependency-ordered before” relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with “dependency-ordered before” followed by “sequenced before”. The reason for this limitation is that a consume operation participating in a “dependency-ordered before” relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of “sequenced before”. The reasons for this limitation are (1) to permit “inter-thread happens before” to be transitively closed and (2) the “happens before” relation, defined below, provides for relationships consisting entirely of “sequenced before”. — end note]

10 An evaluation A happens before an evaluation B (or, equivalently, B happens after A) if:

(10.1) — A is sequenced before B, or

(10.2) — A inter-thread happens before B.

The implementation shall ensure that no program execution demonstrates a cycle in the “happens before” relation.

[Note 9: This cycle would otherwise be possible only through the use of consume operations. — end note]

11 An evaluation A simply happens before an evaluation B if either

(11.1) — A is sequenced before B, or

(11.2) — A synchronizes with B, or

(11.3) — A simply happens before X and X simply happens before B.

[Note 10: In the absence of consume operations, the happens before and simply happens before relations are identical. — end note]

12 An evaluation A strongly happens before an evaluation D if, either

(12.1) — A is sequenced before D, or

(12.2) — A synchronizes with D, and both A and D are sequentially consistent atomic operations (31.4), or

(12.3) — there are evaluations B and C such that A is sequenced before B, B simply happens before C, and C is sequenced before D, or

(12.4) — there is an evaluation B such that A strongly happens before B, and B strongly happens before D.

[Note 11: Informally, if A strongly happens before B, then A appears to be evaluated before B in all contexts. Strongly happens before excludes consume operations. — end note]
A visible side effect A on a scalar object or bit-field M with respect to a value computation B of M satisfies the conditions:

(13.1) — A happens before B and

(13.2) — there is no other side effect X to M such that A happens before X and X happens before B.

The value of a non-atomic scalar object or bit-field M, as determined by evaluation B, shall be the value stored by the visible side effect A.

[Note 12: If there is ambiguity about which side effect to a non-atomic object or bit-field is visible, then the behavior is either unspecified or undefined. — end note]

[Note 13: This states that operations on ordinary objects are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined below, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution. — end note]

The value of an atomic object M, as determined by evaluation B, shall be the value stored by some side effect A that modifies M, where B does not happen before A.

[Note 14: The set of such side effects is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below. — end note]

If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M.

[Note 15: This requirement is known as write-write coherence. — end note]

If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M.

[Note 16: This requirement is known as read-read coherence. — end note]

If a value computation A of an atomic object M happens before an operation B that modifies M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M.

[Note 17: This requirement is known as read-write coherence. — end note]

If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M.

[Note 18: This requirement is known as write-read coherence. — end note]

[Note 19: The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. — end note]

[Note 20: The value observed by a load of an atomic depends on the “happens before” relation, which depends on the values observed by loads of atomic. The intended reading is that there must exist an association of atomic load with modifications they observe that, together with suitably chosen modification orders and the “happens before” relation derived as described above, satisfy the resulting constraints as imposed here. — end note]

Two actions are potentially concurrent if

(21.1) — they are performed by different threads, or

(21.2) — they are unsequenced, at least one is performed by a signal handler, and they are not both performed by the same signal handler invocation.

The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior.

[Note 21: It can be shown that programs that correctly use mutexes and memory_order::seq_cst operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as “sequential consistency”. However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result has undefined behavior. — end note]

Two accesses to the same object of type volatile std::sig_atomic_t do not result in a data race if both occur in the same thread, even if one or more occurs in a signal handler. For each signal handler invocation, evaluations performed by the thread invoking a signal handler can be divided into two groups A

§ 6.9.2.2
and $B$, such that no evaluations in $B$ happen before evaluations in $A$, and the evaluations of such `volatile std::sig_atomic_t` objects take values as though all evaluations in $A$ happened before the execution of the signal handler and the execution of the signal handler happened before all evaluations in $B$.

**Note 22**: Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this document, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question might alias is also generally precluded, since this could violate the coherence rules. —end note

**Note 23**: Transformations that introduce a speculative read of a potentially shared memory location might not preserve the semantics of the C++ program as defined in this document, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection. —end note

### 6.9.2.3 Forward progress

The implementation may assume that any thread will eventually do one of the following:

1. **terminate,**
2. **make a call to a library I/O function,**
3. **perform an access through a volatile glvalue,** or
4. **perform a synchronization operation or an atomic operation.**

**Note 1**: This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. —end note

Executions of atomic functions that are either defined to be lock-free (31.10) or indicated as lock-free (31.5) are **lock-free executions**.

1. If there is only one thread that is not blocked (3.7) in a standard library function, a lock-free execution in that thread shall complete.

   **Note 2**: Concurrently executing threads might prevent progress of a lock-free execution. For example, this situation can occur with load-locked store-conditional implementations. This property is sometimes termed obstruction-free. —end note

2. When one or more lock-free executions run concurrently, at least one should complete.

   **Note 3**: It is difficult for some implementations to provide absolute guarantees to this effect, since repeated and particularly inopportune interference from other threads could prevent forward progress, e.g., by repeatedly stealing a cache line for unrelated purposes between load-locked and store-conditional instructions. For implementations that follow this recommendation and ensure that such effects cannot indefinitely delay progress under expected operating conditions, such anomalies can therefore safely be ignored by programmers. Outside this document, this property is sometimes termed lock-free. —end note

During the execution of a thread of execution, each of the following is termed an **execution step**:

1. **termination of the thread of execution,**
2. **performing an access through a volatile glvalue,** or
3. **completion of a call to a library I/O function, a synchronization operation, or an atomic operation.**

An invocation of a standard library function that blocks (3.7) is considered to continuously execute execution steps while waiting for the condition that it blocks on to be satisfied.

**Example 1**: A library I/O function that blocks until the I/O operation is complete can be considered to continuously check whether the operation is complete. Each such check might consist of one or more execution steps, for example using observable behavior of the abstract machine. —end example

5. **Because of this and the preceding requirement regarding what threads of execution have to perform eventually, it follows that no thread of execution can execute forever without an execution step occurring.** —end note

A thread of execution makes progress when an execution step occurs or a lock-free execution does not complete because there are other concurrent threads that are not blocked in a standard library function (see above).

For a thread of execution providing concurrent forward progress guarantees, the implementation ensures that the thread will eventually make progress for as long as it has not terminated.
An implementation should ensure that the last value (in modification order) assigned by an atomic or

For a thread of execution providing parallel forward progress guarantees, the implementation is not required to ensure that the thread will eventually make progress if it has not yet executed any execution step; once this thread has executed a step, it provides concurrent forward progress guarantees.

For a thread of execution providing weakly parallel forward progress guarantees, the implementation does not ensure that the thread will eventually make progress.

Concurrent forward progress guarantees are stronger than parallel forward progress guarantees, which in turn are stronger than weakly parallel forward progress guarantees.

When a thread of execution $P$ is specified to block with forward progress guarantee delegation on the completion of a set $S$ of threads of execution, then throughout the whole time of $P$ being blocked on $S$, the implementation shall ensure that the forward progress guarantees provided by at least one thread of execution in $S$ is at least as strong as $P$’s forward progress guarantees.

Once a thread of execution in $S$ terminates, it is removed from $S$. Once $S$ is empty, $P$ is unblocked.

A program shall contain a global function called main (6.9.3.1) and the threads of execution created by std::thread (32.4.3) or std::jthread (32.4.4) provide concurrent forward progress guarantees. General-purpose implementations should provide these guarantees.

For a thread of execution providing parallel forward progress guarantees, the implementation is not required to ensure that the thread will eventually make progress if it has not yet executed any execution step; once this thread has executed a step, it provides concurrent forward progress guarantees.

It is implementation-defined whether the implementation-created thread of execution that executes main (6.9.3.1) and the threads of execution created by std::thread (32.4.3) or std::jthread (32.4.4) provide concurrent forward progress guarantees. General-purpose implementations should provide these guarantees.

For a thread of execution providing weakly parallel forward progress guarantees, the implementation does not ensure that the thread will eventually make progress.

Concurrent forward progress guarantees are stronger than parallel forward progress guarantees, which in turn are stronger than weakly parallel forward progress guarantees.

When a thread of execution $P$ is specified to block with forward progress guarantee delegation on the completion of a set $S$ of threads of execution, then throughout the whole time of $P$ being blocked on $S$, the implementation shall ensure that the forward progress guarantees provided by at least one thread of execution in $S$ is at least as strong as $P$’s forward progress guarantees.

An implementation should ensure that the last value (in modification order) assigned by an atomic or synchronization operation will become visible to all other threads in a finite period of time.

6.9.3 Start and termination

6.9.3.1 main function

A program shall contain a global function called main attached to the global module. Executing a program starts a main thread of execution (6.9.2, 32.4) in which the main function is invoked. It is implementation-defined whether a program in a freestanding environment is required to define a main function.

[Note 1: In a freestanding environment, startup and termination is implementation-defined; startup contains the execution of constructor for objects of namespace scope with static storage duration; termination contains the execution of destructors for objects with static storage duration. — end note]
An implementation shall not redefine the main function. This function shall not be overloaded. Its type shall have C++ language linkage and it shall have a declared return type of type int, but otherwise its type is implementation-defined. An implementation shall allow both

- a function of () returning int and
- a function of (int, pointer to pointer to char) returning int

as the type of main (9.3.4.6). In the latter form, for purposes of exposition, the first function parameter is called argc and the second function parameter is called argv, where argc shall be the number of arguments passed to the program from the environment in which the program is run. If argc is nonzero these arguments shall be supplied in argv[0] through argv[argc-1] as pointers to the initial characters of null-terminated multibyte strings (ntmbs) (16.3.3.3.5.3) and argv[0] shall be the pointer to the initial character of a ntmbs that represents the name used to invoke the program or "". The value of argc shall be non-negative. The value of argv[argc] shall be 0.

[Note 2: It is recommended that any further (optional) parameters be added after argv. — end note]

The function main shall not be used within a program. The linkage (6.6) of main is implementation-defined. A program that defines main as deleted or that declares main to be inline, static, or constexpr is ill-formed. The function main shall not be a coroutine (9.5.4). The main function shall not be declared with a linkage-specification (9.11). A program that declares a variable main at global scope, or that declares a function main at global scope attached to a named module, or that declares the name main with C language linkage (in any namespace) is ill-formed. The name main is not otherwise reserved.

[Example 1: Member functions, classes, and enumerations can be called main, as can entities in other namespaces. — end example]

Terminating the program without leaving the current block (e.g., by calling the function std::exit(int) (17.5)) does not destroy any objects with automatic storage duration (11.4.7). If std::exit is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.

A return statement (8.7.4) in main has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling std::exit with the return value as the argument. If control flows off the end of the compound-statement of main, the effect is equivalent to a return with operand 0 (see also 14.4).

### 6.9.3.2 Static initialization

Variables with static storage duration are initialized as a consequence of program initiation. Variables with thread storage duration are initialized as a consequence of thread execution. Within each of these phases of initiation, initialization occurs as follows.

- **Constant initialization** is performed if a variable or temporary object with static or thread storage duration is constant-initialized (7.7). If constant initialization is not performed, a variable with static storage duration (6.7.5.2) or thread storage duration (6.7.5.3) is zero-initialized (9.4). Together, zero-initialization and constant initialization are called static initialization; all other initialization is dynamic initialization. All static initialization strongly happens before (6.9.2.2) any dynamic initialization.

[Note 1: The dynamic initialization of non-local variables is described in 6.9.3.3; that of local static variables is described in 8.8. — end note]

An implementation is permitted to perform the initialization of a variable with static or thread storage duration as a static initialization even if such initialization is not required to be done statically, provided that

- the dynamic version of the initialization does not change the value of any other object of static or thread storage duration prior to its initialization, and
- the static version of the initialization produces the same value in the initialized variable as would be produced by the dynamic initialization if all variables not required to be initialized statically were initialized dynamically.

[Note 2: As a consequence, if the initialization of an object obj1 refers to an object obj2 of namespace scope potentially requiring dynamic initialization and defined later in the same translation unit, it is unspecified whether the value of obj2 used will be the value of the fully initialized obj2 (because obj2 was statically initialized) or will be the value of obj2 merely zero-initialized. For example,

    inline double fd() { return 1.0; }
    extern double d1;

    int main(int argc, char **argv) {
        std::exit(d1); // buggy: static initialization of obj1 refers to obj2
    }

    int main() {
        std::exit(fd()); // correct: static initialization of obj2
    }]

§ 6.9.3.2
Dynamic initialization of non-local variables

1 Dynamic initialization of a non-local variable with static storage duration is unordered if the variable is an implicitly or explicitly instantiated specialization, is partially-ordered if the variable is an inline variable that is not an implicitly or explicitly instantiated specialization, and otherwise is ordered.

[Note 1: An explicitly specialized non-inline static data member or variable template specialization has ordered initialization. —end note]

2 A declaration D is appearance-ordered before a declaration E if

(2.1) D appears in the same translation unit as E, or

(2.2) the translation unit containing E has an interface dependency on the translation unit containing D, in either case prior to E.

3 Dynamic initialization of non-local variables V and W with static storage duration are ordered as follows:

(3.1) If V and W have ordered initialization and the definition of V is appearance-ordered before the definition of W, or if V has partially-ordered initialization, W does not have unordered initialization, and for every definition E of W there exists a definition D of V such that D is appearance-ordered before E, then

(3.1.1) if the program does not start a thread (6.9.2) other than the main thread (6.9.3.1) or V and W have ordered initialization and they are defined in the same translation unit, the initialization of V is sequenced before the initialization of W;

(3.1.2) otherwise, the initialization of V strongly happens before the initialization of W.

(3.2) Otherwise, if the program starts a thread other than the main thread before either V or W is initialized, it is unspecified in which threads the initializations of V and W occur; the initializations are unsequenced if they occur in the same thread.

(3.3) Otherwise, the initializations of V and W are indeterminately sequenced.

[Note 2: This definition permits initialization of a sequence of ordered variables concurrently with another sequence. —end note]

4 A non-initialization odr-use is an odr-use (6.3) not caused directly or indirectly by the initialization of a non-local static or thread storage duration variable.

5 It is implementation-defined whether the dynamic initialization of a non-local non-inline variable with static storage duration is sequenced before the first statement of main or is deferred. If it is deferred, it strongly happens before any non-initialization odr-use of any non-inline function or non-inline variable defined in the same translation unit as the variable to be initialized. It is implementation-defined in which threads and at which points in the program such deferred dynamic initialization occurs.

Recommended practice: An implementation should choose such points in a way that allows the programmer to avoid deadlocks.

[Example 1]:

```c
#include "a.h"
#include "b.h"
B b;
A::A(){
  b.Use();
}
```

51) A non-local variable with static storage duration having initialization with side effects is initialized in this case, even if it is not itself odr-used (6.3, 6.7.5.2).
```c
// - File 3 -
#include "a.h"
#include "b.h"
extern A a;
extern B b;

int main() {
    a.Use();
    b.Use();
}
```

It is implementation-defined whether either `a` or `b` is initialized before `main` is entered or whether the initializations are delayed until `a` is first odr-used in `main`. In particular, if `a` is initialized before `main` is entered, it is not guaranteed that `b` will be initialized before it is odr-used by the initialization of `a`, that is, before `A::A` is called. If, however, `a` is initialized at some point after the first statement of `main`, `b` will be initialized prior to its use in `A::A`. — end example]

6 It is implementation-defined whether the dynamic initialization of a non-local inline variable with static storage duration is sequenced before the first statement of `main` or is deferred. If it is deferred, it strongly happens before any non-initialization odr-use of that variable. It is implementation-defined in which threads and at which points in the program such deferred dynamic initialization occurs.

7 It is implementation-defined whether the dynamic initialization of a non-local non-inline variable with thread storage duration is sequenced before the first statement of the initial function of a thread or is deferred. If it is deferred, the initialization associated with the entity for thread `t` is sequenced before the first non-initialization odr-use by `t` of any non-inline variable with thread storage duration defined in the same translation unit as the variable to be initialized. It is implementation-defined in which threads and at which points in the program such deferred dynamic initialization occurs.

8 If the initialization of a non-local variable with static or thread storage duration exits via an exception, the function `std::terminate` is called (14.6.2).

### 6.9.3.4 Termination

1 Constructed objects (9.4) with static storage duration are destroyed and functions registered with `std::atexit` are called as part of a call to `std::exit` (17.5). The call to `std::exit` is sequenced before the destructions and the registered functions.

[Note 1: Returning from `main` invokes `std::exit` (6.9.3.1). — end note]

2 Constructed objects with thread storage duration within a given thread are destroyed as a result of returning from the initial function of that thread and as a result of that thread calling `std::exit`. The destruction of all constructed objects with thread storage duration within that thread strongly happens before destroying any object with static storage duration.

3 If the completion of the constructor or dynamic initialization of an object with static storage duration strongly happens before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. If the completion of the constructor or dynamic initialization of an object with thread storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. If an object is initialized statically, the object is destroyed in the same order as if the object was dynamically initialized. For an object of array or class type, all subobjects of that object are destroyed before any block-scope object with static storage duration initialized during the construction of the subobjects is destroyed. If the destruction of an object with static or thread storage duration exits via an exception, the function `std::terminate` is called (14.6.2).

4 If a function contains a block-scope object of static or thread storage duration that has been destroyed and the function is called during the destruction of an object with static or thread storage duration, the program has undefined behavior if the flow of control passes through the definition of the previously destroyed block-scope object. Likewise, the behavior is undefined if the block-scope object is used indirectly (i.e., through a pointer) after its destruction.

5 If the completion of the initialization of an object with static storage duration strongly happens before a call to `std::atexit` (see `<cstdlib>`, 17.5), the call to the function passed to `std::atexit` is sequenced before the call to the destructor for the object. If a call to `std::atexit` strongly happens before the completion of the initialization of an object with static storage duration, the call to the destructor for the object is sequenced before the call to the function passed to `std::atexit`. If a call to `std::atexit` strongly happens before another call to `std::atexit`, the call to the function passed to the second `std::atexit` call is sequenced before the call to the function passed to the first `std::atexit` call.
If there is a use of a standard library object or function not permitted within signal handlers (17.13) that does not happen before (6.9.2) completion of destruction of objects with static storage duration and execution of `std::atexit` registered functions (17.5), the program has undefined behavior.

[Note 2: If there is a use of an object with static storage duration that does not happen before the object’s destruction, the program has undefined behavior. Terminating every thread before a call to `std::exit` or the exit from `main` is sufficient, but not necessary, to satisfy these requirements. These requirements permit thread managers as static-storage-duration objects. — end note]

Calling the function `std::abort()` declared in `<cstdlib>` (17.2.2) terminates the program without executing any destructors and without calling the functions passed to `std::atexit()` or `std::at_quick_exit()`.
7 Expressions

7.1 Preamble

[Note 1: Clause 7 defines the syntax, order of evaluation, and meaning of expressions.\textsuperscript{52} An expression is a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects. — end note]

[Note 2: Operators can be overloaded, that is, given meaning when applied to expressions of class type (Clause 11) or enumeration type (9.7.1). Uses of overloaded operators are transformed into function calls as described in 12.6. Overloaded operators obey the rules for syntax and evaluation order specified in 7.6, but the requirements of operand type and value category are replaced by the rules for function call. Relations between operators, such as \texttt{++a} meaning \texttt{a+=1}, are not guaranteed for overloaded operators (12.6). — end note]

Subclause 7.6 defines the effects of operators when applied to types for which they have not been overloaded. Operator overloading shall not modify the rules for the built-in operators, that is, for operators applied to types for which they are defined by this Standard. However, these built-in operators participate in overload resolution, and as part of that process user-defined conversions will be considered where necessary to convert the operands to types appropriate for the built-in operator. If a built-in operator is selected, such conversions will be applied to the operands before the operation is considered further according to the rules in subclause 7.6; see 12.4.2.3, 12.7.

If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

[Note 3: Treatment of division by zero, forming a remainder using a zero divisor, and all floating-point exceptions varies among machines, and is sometimes adjustable by a library function. — end note]

[Note 4: The implementation can regroup operators according to the usual mathematical rules only where the operators really are associative or commutative.\textsuperscript{54} For example, in the following fragment

\begin{verbatim}
int a, b;
/* ... */
a = a + 32760 + b + 5;
\end{verbatim}

the expression statement behaves exactly the same as

\begin{verbatim}
a = (((a + 32760) + b) + 5);
\end{verbatim}

due to the associativity and precedence of these operators. Thus, the result of the sum \((a + 32760)\) is next added to \(b\), and that result is then added to 5 which results in the value assigned to \(a\). On a machine in which overflows produce an exception and in which the range of values representable by an \texttt{int} is \([-32768, +32767]\), the implementation cannot rewrite this expression as

\begin{verbatim}
a = (((a + b) + 32765);
\end{verbatim}

since if the values for \(a\) and \(b\) were, respectively, -32754 and -15, the sum \(a + b\) would produce an exception while the original expression would not; nor can the expression be rewritten as either

\begin{verbatim}
a = ((a + 32765) + b);
\end{verbatim}
or

\begin{verbatim}
a = (a + (b + 32765));
\end{verbatim}

since the values for \(a\) and \(b\) might have been, respectively, 4 and -8 or -17 and 12. However on a machine in which overflows do not produce an exception and in which the results of overflows are reversible, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur. — end note]

The values of the floating-point operands and the results of floating-point expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.\textsuperscript{54}]

\textsuperscript{52} The precedence of operators is not directly specified, but it can be derived from the syntax.
\textsuperscript{53} Overloaded operators are never assumed to be associative or commutative.
\textsuperscript{54} The cast and assignment operators must still perform their specific conversions as described in 7.6.1.4, 7.6.3, 7.6.1.9 and 7.6.19.
7.2 Properties of expressions

7.2.1 Value category

Expressions are categorized according to the taxonomy in Figure 1.

![Expression category taxonomy](fig:basic.lval)

1. A glvalue is an expression whose evaluation determines the identity of an object or function.
2. A prvalue is an expression whose evaluation initializes an object or computes the value of an operand of an operator, as specified by the context in which it appears, or an expression that has type `cv void`.
3. An xvalue is a glvalue that denotes an object whose resources can be reused (usually because it is near the end of its lifetime).
4. An lvalue is a glvalue that is not an xvalue.
5. An rvalue is a prvalue or an xvalue.

Every expression belongs to exactly one of the fundamental classifications in this taxonomy: lvalue, xvalue, or prvalue. This property of an expression is called its value category.

[Note 1: The discussion of each built-in operator in 7.6 indicates the category of the value it yields and the value categories of the operands it expects. For example, the built-in assignment operators expect that the left operand is an lvalue and that the right operand is a prvalue and yield an lvalue as the result. User-defined operators are functions, and the categories of values they expect and yield are determined by their parameter and return types. — end note]

[Note 2: Historically, lvalues and rvalues were so-called because they could appear on the left- and right-hand side of an assignment (although this is no longer generally true); glvalues are “generalized” lvalues, prvalues are “pure” rvalues, and xvalues are “eXpiring” lvalues. Despite their names, these terms classify expressions, not values. — end note]

[Note 3: An expression is an xvalue if it is:

- the result of calling a function, whether implicitly or explicitly, whose return type is an rvalue reference to object type (7.6.1.3),
- a cast to an rvalue reference to object type (7.6.1.4, 7.6.1.7, 7.6.1.9, 7.6.1.10, 7.6.1.11, 7.6.3),
- a subscripting operation with an xvalue array operand (7.6.1.2),
- a class member access expression designating a non-static data member of non-reference type in which the object expression is an xvalue (7.6.1.5), or
- a `.*` pointer-to-member expression in which the first operand is an xvalue and the second operand is a pointer to data member (7.6.4).

In general, the effect of this rule is that named rvalue references are treated as lvalues and unnamed rvalue references to objects are treated as xvalues; rvalue references to functions are treated as lvalues whether named or not. — end note]

Example 1:

```c
struct A {
    int m;
};
A& operator+(A, A);
A& f();
A a;
A& ar = static_cast<A&>(a);
```

The expressions `f()`, `f().m`, `static_cast<A&>(a)`, and `a + a` are xvalues. The expression `ar` is an lvalue. — end example]
The result of a glvalue is the entity denoted by the expression. The result of a prvalue is the value that the expression stores into its context; a prvalue that has type `cv void` has no result. A prvalue whose result is the value \( V \) is sometimes said to have or name the value \( V \). The result object of a prvalue is the object initialized by the prvalue; a non-discarded prvalue that has type `cv void` has no result object.

[Note 4: Except when the prvalue is the operand of a `decltype-specifier`, a prvalue of class or array type always has a result object. For a discarded prvalue that has type other than `cv void`, a temporary object is materialized; see 7.2.3. — end note]

Whenever a glvalue appears as an operand of an operator that expects a prvalue for that operand, the lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), or function-to-pointer (7.3.4) standard conversions are applied to convert the expression to a prvalue.

[Note 5: An attempt to bind an rvalue reference to an lvalue is not such a context; see 9.4.4. — end note]

Whenever a prvalue appears as an operand of an operator that expects a glvalue for that operand, the temporary materialization conversion (7.3.5) is applied to convert the expression to an xvalue.

The discussion of reference initialization in 9.4.4 and of temporaries in 6.7.7 indicates the behavior of lvalues and rvalues in other significant contexts.

Unless otherwise indicated (9.2.9.5), a prvalue shall always have complete type or the `void` type; if it has a class type or (possibly multi-dimensional) array of class type, that class shall not be an abstract class (11.7.4). A glvalue shall not have type `cv void`.

[Note 8: A glvalue can have complete or incomplete non-`void` type. Class and array prvalues can have cv-qualified types; other prvalues always have cv-unqualified types. See 7.2.2. — end note]

An lvalue is modifiable unless its type is const-qualified or is a function type.

[Note 9: A program that attempts to modify an object through a nonmodifiable lvalue or through an rvalue is ill-formed (7.6.19, 7.6.1.6, 7.6.2.3). — end note]

If a program attempts to access (3.1) the stored value of an object through a glvalue whose type is not similar (7.3.6) to one of the following types the behavior is undefined:

(11.1) — the dynamic type of the object,
(11.2) — a type that is the signed or unsigned type corresponding to the dynamic type of the object, or
(11.3) — a char, unsigned char, or std::byte type.

If a program invokes a defaulted copy/move constructor or copy/move assignment operator for a union of type \( U \) with a glvalue argument that does not denote an object of type `cv U` within its lifetime, the behavior is undefined.

[Note 10: Unlike in C, C++ has no accesses of class type. — end note]

### 7.2.2 Type

[expr.type]

If an expression initially has the type “reference to \( T \)” (9.3.4.3, 9.4.4), the type is adjusted to \( T \) prior to any further analysis. The expression designates the object or function denoted by the reference, and the expression is an lvalue or an xvalue, depending on the expression.

[Note 1: Before the lifetime of the reference has started or after it has ended, the behavior is undefined (see 6.7.3). — end note]

If a prvalue initially has the type “\( cv \) \( T \)”, where \( T \) is a cv-unqualified non-class, non-array type, the type of the expression is adjusted to \( T \) prior to any further analysis.

The composite pointer type of two operands \( p1 \) and \( p2 \) having types \( T1 \) and \( T2 \), respectively, where at least one is a pointer or pointer-to-member type or std::nullptr_t, is:

(3.1) — if both \( p1 \) and \( p2 \) are null pointer constants, std::nullptr_t;

55) The intent of this list is to specify those circumstances in which an object can or cannot be aliased.
— if either \( p_1 \) or \( p_2 \) is a null pointer constant, \( T_2 \) or \( T_1 \), respectively;

— if \( T_1 \) or \( T_2 \) is “pointer to \( cv \) \( void \)” and the other type is “pointer to \( cv2 \) \( T \)”, where \( T \) is an object type or \( void \), “pointer to \( cv2 void \)”, where \( cv2 \) is the union of \( cv \) and \( cv2 \);

— if \( T_1 \) or \( T_2 \) is “pointer to \textit{noexcept} function” and the other type is “pointer to function”, where the function types are otherwise the same, “pointer to function”;

— if \( T_1 \) or \( T_2 \) is “pointer to \textit{cv1} \( C_1 \)” and \( T_2 \) is “pointer to \textit{cv2} \( C_2 \)”, where \( C_1 \) is reference-related to \( C_2 \) or \( C_2 \) is reference-related to \( C_1 \) (9.4.4), the cv-combined type (7.3.6) of \( T_1 \) and \( T_2 \) or the cv-combined type of \( T_2 \) and \( T_1 \), respectively;

— if \( T_1 \) and \( T_2 \) are similar types (7.3.6), the cv-combined type of \( T_1 \) and \( T_2 \);

— otherwise, a program that necessitates the determination of a composite pointer type is ill-formed.

Example 1:

```c
typedef void *p;
typedef const int *q;
typedef int **pi;
typedef const int **pci;
```

The composite pointer type of \( p \) and \( q \) is “pointer to \textit{const void}”; the composite pointer type of \( pi \) and \( pci \) is “pointer to \textit{const pointer to const int}”. —end example

### 7.2.3 Context dependence

In some contexts, unevaluated operands appear (7.5.7, 7.6.1.8, 7.6.2.5, 7.6.2.7, 9.2.9.5, 13.1, 13.7.9). An unevaluated operand is not evaluated.

**Note 1:** In an unevaluated operand, a non-static class member can be named (7.5.4) and naming of objects or functions does not, by itself, require that a definition be provided (6.3). An unevaluated operand is considered a full-expression (6.9.1). —end note

In some contexts, an expression only appears for its side effects. Such an expression is called a discarded-value expression. The array-to-pointer (7.3.3) and function-to-pointer (7.3.4) standard conversions are not applied. The lvalue-to-rvalue conversion (7.3.2) is applied if and only if the expression is a glvalue of volatile-qualified type and it is one of the following:

1. (expression), where expression is one of these expressions,
2. id-expression (7.5.4),
3. subscripting (7.6.1.2),
4. class member access (7.6.1.5),
5. indirection (7.6.2.2),
6. pointer-to-member operation (7.6.4),
7. conditional expression (7.6.16) where both the second and the third operands are one of these expressions, or
8. comma expression (7.6.20) where the right operand is one of these expressions.

**Note 2:** Using an overloaded operator causes a function call; the above covers only operators with built-in meaning. —end note

If the (possibly converted) expression is a prvalue, the temporary materialization conversion (7.3.5) is applied.

**Note 3:** If the expression is an lvalue of class type, it must have a volatile copy constructor to initialize the temporary object that is the result object of the lvalue-to-rvalue conversion. —end note

The glvalue expression is evaluated and its value is discarded.
7.3 Standard conversions

7.3.1 General

Standard conversions are implicit conversions with built-in meaning. 7.3 enumerates the full set of such conversions. A standard conversion sequence is a sequence of standard conversions in the following order:

1. Zero or one conversion from the following set: lvalue-to-rvalue conversion, array-to-pointer conversion, and function-to-pointer conversion.
2. Zero or one conversion from the following set: integral promotions, floating-point promotion, integral conversions, floating-point conversions, floating-integral conversions, pointer conversions, pointer-to-member conversions, and boolean conversions.
3. Zero or one function pointer conversion.
4. Zero or one qualification conversion.

[Note 1: A standard conversion sequence can be empty, i.e., it can consist of no conversions. —end note]

A standard conversion sequence will be applied to an expression if necessary to convert it to a required destination type.

2 [Note 2: Expressions with a given type will be implicitly converted to other types in several contexts:

1. When used as operands of operators. The operator’s requirements for its operands dictate the destination type (7.6).
2. When used in the condition of an if statement (8.5.2) or iteration statement (8.6). The destination type is bool.
3. When used in the expression of a switch statement (8.5.3). The destination type is integral.
4. When used as the source expression for an initialization (which includes use as an argument in a function call and use as the expression in a return statement). The type of the entity being initialized is (generally) the destination type. See 9.4, 9.4.4.

—end note]

An expression $E$ can be implicitly converted to a type $T$ if and only if the declaration $T t=E$; is well-formed, for some invented temporary variable $t$ (9.4).

4 Certain language constructs require that an expression be converted to a Boolean value. An expression $E$ appearing in such a context is said to be contextually converted to bool and is well-formed if and only if the declaration bool $t(E)$; is well-formed, for some invented temporary variable $t$ (9.4).

5 Certain language constructs require conversion to a value having one of a specified set of types appropriate to the construct. An expression $E$ of class type $C$ appearing in such a context is said to be contextually implicitly converted to a specified type $T$ and is well-formed if and only if $E$ can be implicitly converted to a type $T$ that is determined as follows: $C$ is searched for non-explicit conversion functions whose return type is $T$ or reference to $T$ such that $T$ is allowed by the context. There shall be exactly one such $T$.

6 The effect of any implicit conversion is the same as performing the corresponding declaration and initialization and then using the temporary variable as the result of the conversion. The result is an lvalue if $T$ is an lvalue reference type or an rvalue reference to function type (9.3.4.3), an xvalue if $T$ is an rvalue reference to object type, and a prvalue otherwise. The expression $E$ is used as a gvalue if and only if the initialization uses it as a gvalue.

7 [Note 3: For class types, user-defined conversions are considered as well; see 11.4.8. In general, an implicit conversion sequence (12.4.4.2) consists of a standard conversion sequence followed by a user-defined conversion followed by another standard conversion sequence. —end note]

8 [Note 4: There are some contexts where certain conversions are suppressed. For example, the lvalue-to-rvalue conversion is not done on the operand of the unary & operator. Specific exceptions are given in the descriptions of those operators and contexts. —end note]

7.3.2 Lvalue-to-rvalue conversion

A gvalue (7.2.1) of a non-function, non-array type $T$ can be converted to a prvalue. If $T$ is an incomplete type, a program that necessitates this conversion is ill-formed. If $T$ is a non-class type, the type of the prvalue

56) For historical reasons, this conversion is called the “lvalue-to-rvalue” conversion, even though that name does not accurately reflect the taxonomy of expressions described in 7.2.1.
is the cv-unqualified version of \( T \). Otherwise, the type of the prvalue is \( T \)^57

When an lvalue-to-rvalue conversion is applied to an expression \( E \), and either

1. \( E \) is not potentially evaluated, or
2. the evaluation of \( E \) results in the evaluation of a member \( E_x \) of the set of potential results of \( E \), and \( E_x \) names a variable \( x \) that is not odr-used by \( E_x \) (6.3),

the value contained in the referenced object is not accessed.

**Example 1:**

```cpp
struct S { int n; }
auto f() {
    S x { 1 };
    constexpr S y { 2 };
    return [&] (bool b) { return (b ? y : x).n; }
}
auto g = f();
int m = g(false); // undefined behavior: access of x.n outside its lifetime
int n = g(true);  // OK, does not access y.n
```

3 The result of the conversion is determined according to the following rules:

1. If \( T \) is \( cv \) \( std::nullptr_t \), the result is a null pointer constant (7.3.12).

   **Note 1:** Since the conversion does not access the object to which the glvalue refers, there is no side effect even if \( T \) is volatile-qualified (6.9.1), and the glvalue can refer to an inactive member of a union (11.5). **— end note**

2. Otherwise, if \( T \) has a class type, the conversion copy-initializes the result object from the glvalue.

3. Otherwise, if the object to which the glvalue refers contains an invalid pointer value (6.7.5.5.3, 6.7.5.5.4), the behavior is implementation-defined.

4 Otherwise, the object indicated by the glvalue is read (3.1), and the value contained in the object is the prvalue result.

**Note 2:** See also 7.2.1. **— end note**

### 7.3.3 Array-to-pointer conversion

An lvalue or rvalue of type “array of \( N \ T \)” or “array of unknown bound of \( T \)” can be converted to a prvalue of type “pointer to \( T \).” The temporary materialization conversion (7.3.5) is applied. The result is a pointer to the first element of the array.

### 7.3.4 Function-to-pointer conversion

An lvalue of function type \( T \) can be converted to a prvalue of type “pointer to \( T \).” The result is a pointer to the function.^58

### 7.3.5 Temporary materialization conversion

A prvalue of type \( T \) can be converted to an xvalue of type \( T \). This conversion initializes a temporary object (6.7.7) of type \( T \) from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object. \( T \) shall be a complete type.

**Note 1:** If \( T \) is a class type (or array thereof), it must have an accessible and non-deleted destructor; see 11.4.7. **— end note**

**Example 1:**

```cpp
struct X { int n; }
int k = X().n; // OK, X() prvalue is converted to xvalue
```

57) In C++ class and array prvalues can have cv-qualified types. This differs from ISO C, in which non-lvalues never have cv-qualified types.

58) This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.
7.3.6 Qualification conversions [conv.qual]

1 A cv-decomposition of a type T is a sequence of cv_i and P_i such that T is

\[ \text{"cv}_0 \; P_0 \; \text{cv}_1 \; P_1 \; \cdots \; \text{cv}_{n-1} \; P_{n-1} \; \text{cv}_n \; \text{U} \] \text{ for } n \geq 0,

where each cv_i is a set of cv-qualifiers (6.8.4), and each P_i is “pointer to” (9.3.4.2), “pointer to member of class C_i of type” (9.3.4.4), “array of N_i”, or “array of unknown bound of” (9.3.4.5). If P_i designates an array, the cv-qualifiers cv_{i+1} on the element type are also taken as the cv-qualifiers cv_i of the array.

[Example 1: The type denoted by the type-id \texttt{const int \* \*} has three cv-decompositions, taking U as “int”, as “pointer to \texttt{const int}”, and as “pointer to pointer to \texttt{const int}”. — end example]

The n-tuple of cv-qualifiers after the first one in the longest cv-decomposition of T, that is, cv_1, cv_2, \ldots, cv_n, is called the cv-qualification signature of T.

2 Two types T1 and T2 are similar if they have cv-decompositions with the same n such that corresponding P_i components are either the same or one is “array of N_i” and the other is “array of unknown bound of”, and the types denoted by U are the same.

3 The cv-combined type of two types T1 and T2 is the type T3 similar to T1 whose cv-decomposition is such that:

\[ \begin{align*}
& (3.1) & \text{— for each } i > 0, \text{ cv}_i^1 \text{ is the union of } \text{cv}_i^1 \text{ and } \text{cv}_i^2, \\
& (3.2) & \text{— if either } P_i^1 \text{ or } P_i^2 \text{ is “array of unknown bound of”, } P_i^3 \text{ is “array of unknown bound of”, otherwise it is } P_i^1 \text{ and } P_i^2, \\
& (3.3) & \text{— if the resulting } \text{cv}_i^3 \text{ is different from } \text{cv}_i^1 \text{ or } \text{cv}_i^2, \text{ or the resulting } P_i^3 \text{ is different from } P_i^1 \text{ or } P_i^2, \text{ then cv} \text{ is added to every } \text{cv}_k^i \text{ for } 0 < k < i,
\end{align*} \]

where \text{cv}_i^1 and \text{cv}_i^2 are the components of the cv-decomposition of T_j. A prvalue of type T1 can be converted to type T2 if the cv-combined type of T1 and T2 is T2.

[Note 1: If a program could assign a pointer of type T** to a pointer of type \texttt{const T**} (that is, if line \#1 below were allowed), a program could inadvertently modify a const object (as it is done on line \#2). For example,]

\begin{verbatim}
int main() {
    const char c = 'c';
    char* pc;
    const char** pcc = &pc;       // \#1: not allowed
    *pcc = &c;
    *pc = 'C';                     // \#2: modifies a const object
}
\end{verbatim}

— end note]

[Note 2: Given similar types T1 and T2, this construction ensures that both can be converted to the cv-combined type of T1 and T2. — end note]

4 [Note 3: A prvalue of type “pointer to cv1 T" can be converted to a prvalue of type “pointer to cv2 T" if “cv2 T" is more cv-qualified than “cv1 T". A prvalue of type “pointer to member of X of type cv1 T" can be converted to a prvalue of type “pointer to member of X of type cv2 T" if “cv2 T" is more cv-qualified than “cv1 T". — end note]

5 [Note 4: Function types (including those used in pointer-to-member-function types) are never cv-qualified (9.3.4.6). — end note]

7.3.7 Integral promotions [conv.prom]

1 A prvalue of an integer type other than bool, char16_t, char32_t, or wchar_t whose integer conversion rank (6.8.5) is less than the rank of int can be converted to a prvalue of type int if int can represent all the values of the source type; otherwise, the source prvalue can be converted to a prvalue of type unsigned int.

2 A prvalue of type char16_t, char32_t, or wchar_t (6.8.2) can be converted to a prvalue of the first of the following types that can represent all the values of its underlying type: int, unsigned int, long int, unsigned long int, long long int, or unsigned long long int. If none of the types in that list can represent all the values of its underlying type, a prvalue of type char16_t, char32_t, or wchar_t can be converted to a prvalue of its underlying type.

3 A prvalue of an unscoped enumeration type whose underlying type is not fixed can be converted to a prvalue of the first of the following types that can represent all the values of the enumeration (9.7.1): int, unsigned int, long int, unsigned long int, long long int, or unsigned long long int. If none of the types in that list can represent all the values of the enumeration, a prvalue of an unscoped enumeration type can be
converted to a prvalue of the extended integer type with lowest integer conversion rank (6.8.5) greater than the rank of `long long` in which all the values of the enumeration can be represented. If there are two such extended types, the signed one is chosen.

4 A prvalue of an unscoped enumeration type whose underlying type is fixed (9.7.1) can be converted to a prvalue of its underlying type. Moreover, if integral promotion can be applied to its underlying type, a prvalue of an unscoped enumeration type whose underlying type is fixed can also be converted to a prvalue of the promoted underlying type.

5 A prvalue for an integral bit-field (11.4.10) can be converted to a prvalue of type `int` if `int` can represent all the values of the bit-field; otherwise, it can be converted to `unsigned int` if `unsigned int` can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.

6 A prvalue of type `bool` can be converted to a prvalue of type `int`, with `false` becoming zero and `true` becoming one.

7 These conversions are called *integral promotions*.

7.3.8 Floating-point promotion

1 A prvalue of type `float` can be converted to a prvalue of type `double`. The value is unchanged.

2 This conversion is called *floating-point promotion*.

7.3.9 Integral conversions

1 A prvalue of an integer type can be converted to a prvalue of another integer type. A prvalue of an unscoped enumeration type can be converted to a prvalue of an integer type.

2 If the destination type is `bool`, see 7.3.15. If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

3 Otherwise, the result is the unique value of the destination type that is congruent to the source integer modulo $2^N$, where $N$ is the width of the destination type.

4 The conversions allowed as integral promotions are excluded from the set of integral conversions.

7.3.10 Floating-point conversions

1 A prvalue of floating-point type can be converted to a prvalue of another floating-point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.

2 The conversions allowed as floating-point promotions are excluded from the set of floating-point conversions.

7.3.11 Floating-integral conversions

1 A prvalue of a floating-point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type.

[Note 1: If the destination type is `bool`, see 7.3.15. — end note]

2 A prvalue of an integer type or of an unscoped enumeration type can be converted to a prvalue of a floating-point type. The result is exact if possible. If the value being converted is in the range of values that can be represented but the value cannot be represented exactly, it is an implementation-defined choice of either the next lower or higher representable value.

[Note 2: Loss of precision occurs if the integral value cannot be represented exactly as a value of the floating-point type. — end note]

If the value being converted is outside the range of values that can be represented, the behavior is undefined. If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

7.3.12 Pointer conversions

1 A *null pointer constant* is an integer literal (5.13.2) with value zero or a prvalue of type `std::nullptr_t`. A null pointer constant can be converted to a pointer type; the result is the null pointer value of that type (6.8.3) and is distinguishable from every other value of object pointer or function pointer type. Such a conversion is called a *null pointer conversion*. Two null pointer values of the same type shall compare equal. The
conversion of a null pointer constant to a pointer to cv-qualified type is a single conversion, and not the sequence of a pointer conversion followed by a qualification conversion (7.3.6). A null pointer constant of integral type can be converted to a prvalue of type `std::nullptr_t`.

[Note 1: The resulting prvalue is not a null pointer value. — end note]

2 A prvalue of type “pointer to cv T”, where T is an object type, can be converted to a prvalue of type “pointer to cv void”. The pointer value (6.8.3) is unchanged by this conversion.

3 A prvalue of type “pointer to cv D”, where D is a complete class type, can be converted to a prvalue of type “pointer to cv B”, where B is a base class (11.7) of D. If B is an inaccessible (11.9) or ambiguous (11.8) base class of D, a program that necessitates this conversion is ill-formed. The result of the conversion is a pointer to the base class subobject of the derived class object. The null pointer value is converted to the null pointer value of the destination type.

7.3.13 Pointer-to-member conversions [conv.mem]

1 A null pointer constant (7.3.12) can be converted to a pointer-to-member type; the result is the null member pointer value of that type and is distinguishable from any pointer to member not created from a null pointer constant. Such a conversion is called a null member pointer conversion. Two null member pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to member of cv-qualified type is a single conversion, and not the sequence of a pointer-to-member conversion followed by a qualification conversion (7.3.6).

2 A prvalue of type “pointer to member of B of type cv T”, where B is a class type, can be converted to a prvalue of type “pointer to member of D of type cv T”, where D is a complete class derived (11.7) from B. If B is an inaccessible (11.9), ambiguous (11.8), or virtual (11.7.2) base class of D, or a base class of a virtual base class of D, a program that necessitates this conversion is ill-formed. The result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The result refers to the member in D’s instance of B. Since the result has type “pointer to member of D of type cv T”, indirection through it with a D object is valid. The result is the same as if indirection through the pointer to member of B with the B subobject of D. The null member pointer value is converted to the null member pointer value of the destination type.59

7.3.14 Function pointer conversions [conv.fctptr]

1 A prvalue of type “pointer to noexcept function” can be converted to a prvalue of type “pointer to function”. The result is a pointer to the function. A prvalue of type “pointer to member of type noexcept function” can be converted to a prvalue of type “pointer to member of type function”. The result designates the member function.

[Example 1:
void (p)();
void (**pp)() noexcept = &p; // error: cannot convert to pointer to noexcept function

struct S { typedef void (p)(); operator p(); };
void (*q)() noexcept = S(); // error: cannot convert to pointer to noexcept function
— end example]

7.3.15 Boolean conversions [conv.bool]

1 A prvalue of arithmetic, unscoped enumeration, pointer, or pointer-to-member type can be converted to a prvalue of type `bool`. A zero value, null pointer value, or null member pointer value is converted to `false`; any other value is converted to `true`.

7.4 Usual arithmetic conversions [expr.arith.conv]

1 Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions, which are defined as follows:

§ 7.4 99
If either operand is of scoped enumeration type (9.7.1), no conversions are performed; if the other operand does not have the same type, the expression is ill-formed.

If either operand is of type `long double`, the other shall be converted to `long double`.

Otherwise, if either operand is `double`, the other shall be converted to `double`.

Otherwise, if either operand is `float`, the other shall be converted to `float`.

Otherwise, the integral promotions (7.3.7) shall be performed on both operands.

Then the following rules shall be applied to the promoted operands:

- If both operands have the same type, no further conversion is needed.
- Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank shall be converted to the type of the operand with greater rank.
- Otherwise, if the type of the operand with signed integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type shall be converted to the type of the operand with unsigned integer type.
- Otherwise, if the type of the operand with unsigned integer type can represent all of the values of the type of the operand with signed integer type, the operand with unsigned integer type shall be converted to the type of the operand with signed integer type.
- Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

If one operand is of enumeration type and the other operand is of a different enumeration type or a floating-point type, this behavior is deprecated (D.2).

### 7.5 Primary expressions

```plaintext
primary-expression:
  literal
  this
  ( expression )
  id-expression
  lambda-expression
  fold-expression
  requires-expression
```

#### 7.5.1 Literals

A `literal` is a primary expression. The type of a `literal` is determined based on its form as specified in 5.13. A `string-literal` is an lvalue, a `user-defined-literal` has the same value category as the corresponding operator call expression described in 5.13.8, and any other `literal` is a prvalue.

#### 7.5.2 This

The keyword `this` names a pointer to the object for which a non-static member function (11.4.3.2) is invoked or a non-static data member’s initializer (11.4) is evaluated.

If a declaration declares a member function or member function template of a class X, the expression `this` is a prvalue of type “pointer to cv-qualifier-seq X” between the optional cv-qualifier-seq and the end of the function-definition, member-declarator, or declarator. It shall not appear before the optional cv-qualifier-seq and it shall not appear within the declaration of a static member function (although its type and value category are defined within a static member function as they are within a non-static member function).

[Note 1: This is because declaration matching does not occur until the complete declarator is known. — end note]

[Note 2: In a trailing-return-type, the class being defined is not required to be complete for purposes of class member access (7.6.1.5). Class members declared later are not visible.]
```cpp
template<class T> auto f(T t) -> decltype(t + g())
{ return t + g(); }
}
template auto A::f(int t) -> decltype(t + g());
— end example
— end note

Otherwise, if a member-declarator declares a non-static data member (11.4) of a class \( X \), the expression \texttt{this} is a prvalue of type “pointer to \( X \)” within the optional default member initializer (11.4). It shall not appear elsewhere in the member-declarator.

The expression \texttt{this} shall not appear in any other context.

[Example 2:]
```
```cpp
class Outer {
  int a[sizeof(*this)]; // error: not inside a member function
  unsigned int sz = sizeof(*this); // OK: in default member initializer

  void f() {
    int b[sizeof(*this)]; // OK
    struct Inner {
      int c[sizeof(*this)]; // error: not inside a member function of Inner
    }
  }
};
— end example]

7.5.3 Parentheses

A parenthesized expression \((E)\) is a primary expression whose type, result, and value category are identical to those of \( E \). The parenthesized expression can be used in exactly the same contexts as those where \( E \) can be used, and with the same meaning, except as otherwise indicated.

7.5.4 Names

7.5.4.1 General

id-expression:
  unqualified-id
  qualified-id

1 An id-expression is a restricted form of a primary-expression.

[Note 1: An id-expression can appear after . and -> operators (7.6.1.5). — end note]

2 An id-expression that denotes a non-static data member or non-static member function of a class can only be used:

(2.1) — as part of a class member access (7.6.1.5) in which the object expression refers to the member’s class\(^{61}\) or a class derived from that class, or

(2.2) — to form a pointer to member (7.6.2.2), or

(2.3) — if that id-expression denotes a non-static data member and it appears in an unevaluated operand.

[Example 1:]
```cpp
struct S {
  int m;
};
int i = sizeof(S::m); // OK
int j = sizeof(S::m + 42); // OK
— end example]

3 A potentially-evaluated id-expression that denotes an immediate function (9.2.6) shall appear only

(3.1) — as a subexpression of an immediate invocation, or

\(^{61}\) This also applies when the object expression is an implicit \((\texttt{*this})\) (11.4.3).
For an id-expression that denotes an overload set, overload resolution is performed to select a unique function (12.4, 12.5).

[Note 2: A program cannot refer to a function with a trailing requires-clause whose constraint-expression is not satisfied, because such functions are never selected by overload resolution.]

[Example 2:

```cpp
template<typename T> struct A {
    static void f(int) requires false;
};

void g() {
    A<int>::f(0); // error: cannot call f
    void (*p1)(int) = A<int>::f; // error: cannot take the address of f
degltype(A<int>::f)* p2 = nullptr; // error: the type decltype(A<int>::f) is invalid
}
```

In each case, the constraints of f are not satisfied. In the declaration of p2, those constraints are required to be satisfied even though f is an unevaluated operand (7.2). — end example]

— end note]

7.5.4.2 Unqualified names

unqualified-id:

identifier
operator-function-id
conversion-function-id
literal-operator-id
~ type-name
~ decltype-specifier
template-id

An identifier is only an id-expression if it has been suitably declared (Clause 9) or if it appears as part of a declarator-id (9.3). An identifier that names a coroutine parameter refers to the copy of the parameter (9.5.4).

[Note 1: For operator-function-ids, see 12.6; for conversion-function-ids, see 11.4.8.3; for literal-operator-ids, see 12.8; for template-ids, see 13.3. A type-name or decltype-specifier prefixed by ~ denotes the destructor of the type so named; see 7.5.4.4. Within the definition of a non-static member function, an identifier that names a non-static member is transformed to a class member access expression (11.4.3). — end note]

The result is the entity denoted by the identifier. If the entity is a local entity and naming it from outside of an unevaluated operand within the declarative region where the unqualified-id appears would result in some intervening lambda-expression capturing it by copy (7.5.5.3), the type of the expression is the type of a class member access expression (7.6.1.5) naming the non-static data member that would be declared for such a capture in the closure object of the innermost such intervening lambda-expression.

[Note 2: If that lambda-expression is not declared mutable, the type of such an identifier will typically be const qualified. — end note]

Otherwise, the type of the expression is the type of the result.

[Note 3: If the entity is a template parameter object for a template parameter of type T (13.2), the type of the expression is const T. — end note]

The expression is an lvalue if the entity is a function, variable, structured binding (9.6), data member, or template parameter object and a prvalue otherwise (7.2.1); it is a bit-field if the identifier designates a bit-field.

[Example 1:

```cpp
void f() {
    float x, &r = x;
    [=] {
        decltype(x) y1;
        decltype(x) y2 = y1; // y2 has type float
        decltype(r) r1 = y1; // r1 has type float
    }
```
7.5.4.3 Qualified names

qualified-id:
  nested-name-specifier template_opt unqualified-id

nested-name-specifier:
  ::
  type-name ::
  namespace-name ::
  decltype-specifier ::
  nested-name-specifier identifier ::
  nested-name-specifier template_opt simple-template-id ::

1. The type denoted by a decltype-specifier in a nested-name-specifier shall be a class or enumeration type.
2. A nested-name-specifier that denotes a class, optionally followed by the keyword template (13.3), and then followed by the name of a member of either that class (11.4) or one of its base classes (11.7), is a qualified-id; 6.5.4.2 describes name lookup for class members that appear in qualified-ids. The result is the member. The type of the result is the type of the member. The result is an lvalue if the member is a static member function or a data member and a prvalue otherwise.

[Note 1: A class member can be referred to using a qualified-id at any point in its potential scope (6.4.7). — end note]

Where type-name :: ~ type-name is used, the two type-names shall refer to the same type (ignoring cv-qualifications); this notation denotes the destructor of the type so named (7.5.4.4). The unqualified-id in a qualified-id shall not be of the form ~decltype-specifier.

3. The nested-name-specifier :: names the global namespace. A nested-name-specifier that names a namespace (9.8), optionally followed by the keyword template (13.3), and then followed by the name of a member of that namespace (or the name of a member of a namespace made visible by a using-directive), is a qualified-id; 6.5.4.3 describes name lookup for namespace members that appear in qualified-ids. The result is the member. The type of the result is the type of the member. The result is an lvalue if the member is a function, a variable, or a structured binding (9.6) and a prvalue otherwise.

4. A nested-name-specifier that denotes an enumeration (9.7.1), followed by the name of an enumerator of that enumeration, is a qualified-id that refers to the enumerator. The result is the enumerator. The type of the result is the type of the enumeration. The result is a prvalue.

5. In a qualified-id, if the unqualified-id is a conversion-function-id, its conversion-type-id is first looked up in the class denoted by the nested-name-specifier of the qualified-id and the name, if found, is used. Otherwise, it is looked up in the context in which the entire qualified-id occurs. In each of these lookups, only names that denote types or templates whose specializations are types are considered.

7.5.4.4 Destruction

An id-expression that denotes the destructor of a type T names the destructor of T if T is a class type (11.4.7), otherwise the id-expression is said to name a pseudo-destructor.

If the id-expression names a pseudo-destructor, T shall be a scalar type and the id-expression shall appear as the right operand of a class member access (7.6.1.5) that forms the postfix-expression of a function call (7.6.1.3).

[Note 1: Such a call ends the lifetime of the object (7.6.1.3, 6.7.3). — end note]

3. [Example 1:]

```c
struct C { }
void f() {
  C * pc = new C;
  using C2 = C;
  pc->C:: ~C2(); // OK, destroys *pc
  C().C:: ~C(); // undefined behavior: temporary of type C destroyed twice
  using T = int;
  0 .T:: ~T(); // OK, no effect
  0.T:: ~T(); // error: 0.T is a user-defined-floating-point-literal (5.13.8)
}
```
7.5.5 Lambda expressions

7.5.5.1 General

A lambda-expression provides a concise way to create a simple function object.

[Example 1]:
```cpp
#include <algorithm>
#include <cmath>
void abssort(float* x, unsigned N) {
    std::sort(x, x + N, [](float a, float b) { return std::abs(a) < std::abs(b); });
}
```

— end example]

1 A lambda-expression is a prvalue whose result object is called the closure object.

[Note 1: A closure object behaves like a function object (20.14). — end note]

2 In the decl-specifier-seq of the lambda-declarator, each decl-specifier shall be one of mutable, constexpr, or consteval.

[Note 2: The trailing requires-clause is described in 9.3. — end note]

3 If a lambda-expression does not include a lambda-declarator, it is as if the lambda-declarator were (). The lambda return type is auto, which is replaced by the type specified by the trailing-return-type if provided and/or deduced from return statements as described in 9.2.9.6.

[Example 2]:
```cpp
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []( return { 1, 2 }; ); // error: deducing return type from braced-init-list
int j;
auto x3 = [](()->auto&& { return j; }); // OK: return type is int&
```

— end example]

4 A lambda is a generic lambda if the lambda-expression has any generic parameter type placeholders (9.2.9.6), or if the lambda has a template-parameter-list.

[Example 3]:
```cpp
int i = [](int i, auto a) { return i; }(3, 4); // OK: a generic lambda
int j = [](<class T>(T t, int i) { return i; })(3, 4); // OK: a generic lambda
```

— end example]

7.5.5.2 Closure types

1 The type of a lambda-expression (which is also the type of the closure object) is a unique, unnamed non-union class type, called the closure type, whose properties are described below.

2 The closure type is declared in the smallest block scope, class scope, or namespace scope that contains the corresponding lambda-expression.

[Note 1: This determines the set of namespaces and classes associated with the closure type (6.5.3). The parameter types of a lambda-declarator do not affect these associated namespaces and classes. — end note]

The closure type is not an aggregate type (9.4.2). An implementation may define the closure type differently from what is described below provided this does not alter the observable behavior of the program other than by changing:

(2.1) — the size and/or alignment of the closure type,
whether the closure type is trivially copyable (11.2), or
whether the closure type is a standard-layout class (11.2).

An implementation shall not add members of rvalue reference type to the closure type.

3 The closure type for a lambda-expression has a public inline function call operator (for a non-generic lambda) or function call operator template (for a generic lambda) (12.6.4) whose parameters and return type are described by the lambda-expression’s parameter-declaration-clause and trailing-return-type respectively, and whose template-parameter-list consists of the specified template-parameter-list, if any. The requires-clause of the function call operator template is the requires-clause immediately following < template-parameter-list >, if any. The trailing requires-clause of the function call operator or operator template is the requires-clause of the lambda-declarator, if any.

[Note 2: The function call operator template for a generic lambda might be an abbreviated function template (9.3.4.6). —end note]

[Example 1:

```cpp
auto glambda = [](auto a, auto&& b) { return a < b; };
bool b = glambda(3, 3.14); // OK

auto vglambda = [](auto printer) {
    return [=](auto&& ... ts) {
        printer(std::forward<decltype(ts)>(ts)...); // OK: ts is a function parameter pack
        return [=]() {
            printer(ts ...);
        };
    };
}

auto p = vglambda( [](auto v1, auto v2, auto v3) {
    std::cout << v1 << v2 << v3; } );

auto q = p(1, 'a', 3.14); // OK: outputs 1a3.14
q(); // OK: outputs 1a3.14
@end example]

4 The function call operator or operator template is declared const (11.4.3) if and only if the lambda-expression’s parameter-declaration-clause is not followed by mutable. It is neither virtual nor declared volatile. Any noexcept-specifier specified on a lambda-expression applies to the corresponding function call operator or operator template. An attribute-specifier-seq in a lambda-declarator appertains to the type of the corresponding function call operator or operator template. The function call operator or any given operator template specialization is a constexpr function if either the corresponding lambda-expression’s parameter-declaration-clause is followed by constexpr or consteval, or it satisfies the requirements for a constexpr function (9.2.6). It is an immediate function (9.2.6) if the corresponding lambda-expression’s parameter-declaration-clause is followed by consteval.

[Note 3: Names referenced in the lambda-declarator are looked up in the context in which the lambda-expression appears. —end note]

[Example 2:

```cpp
auto ID = [](auto a) { return a; };
static_assert(ID(3) == 3); // OK

struct NonLiteral {
    NonLiteral(int n) : n(n) {}  
    int n;
};
static_assert(ID(NonLiteral{3}).n == 3); // error
@end example]

5 [Example 3:

```cpp
auto monoid = [](auto v) { return [=] { return v; }; };
auto add = [](auto m1) constexpr {
    auto ret = m1();
    return [=](auto m2) mutable {
        auto m1val = m1();
```
auto plus = [=](auto m2val) mutable constexpr
    { return m1val += m2val; }
ret = plus(m2());
return monoid(ret);
};
constexpr auto zero = monoid(0);
constexpr auto one = monoid(1);
static_assert(add(one)(zero)() == one()); // OK

// Since two below is not declared constexpr, an evaluation of its constexpr member function call operator
// cannot perform an lvalue-to-rvalue conversion on one of its subobjects (that represents its capture)
// in a constant expression.
auto two = monoid(2);
assert(two() == 2); // OK, not a constant expression.
static_assert(add(one)(one)() == two()); // error: two() is not a constant expression
static_assert(add(one)(one)() == monoid(2)()); // OK
—end example

[Note 4: The function call operator or operator template can be constrained (13.5.3) by a type-constraint (13.2), a
requires-clause (13.1), or a trailing requires-clause (9.3).

[Example 4:

    template <typename T> concept C1 = /* ... */;
    template <std::size_t N> concept C2 = /* ... */;
    template <typename A, typename B> concept C3 = /* ... */;

    auto f = []<typename T1, C1 T2> requires C2<sizeof(T1) + sizeof(T2)>
        (T1 a1, T1 b1, T2 a2, auto a3, auto a4) requires C3<decltype(a4), T2>
        { // T2 is constrained by a type-constraint.
        // T1 and T2 are constrained by a requires-clause, and
        // T2 and the type of a4 are constrained by a trailing requires-clause.
        };
—end example]
—end note]

The closure type for a non-generic lambda-expression with no lambda-capture whose constraints (if any) are
satisfied has a conversion function to pointer to function with C++ language linkage (9.11) having the same
parameter and return types as the closure type’s function call operator. The conversion is to “pointer to
noexcept function” if the function call operator has a non-throwing exception specification. The value
returned by this conversion function is the address of a function F that, when invoked, has the same effect as
invoking the closure type’s function call operator on a default-constructed instance of the closure type. F is a
constexpr function if the function call operator is a constexpr function and is an immediate function if the
function call operator is an immediate function.

For a generic lambda with no lambda-capture, the closure type has a conversion function template to pointer
to function. The conversion function template has the same invented template parameter list, and the pointer
to function has the same parameter types, as the function call operator template. The return type of the
pointer to function shall behave as if it were a decltype-specifier denoting the return type of the corresponding
function call operator template specialization.

[Note 5: If the generic lambda has no trailing-return-type or the trailing-return-type contains a placeholder type, return
type deduction of the corresponding function call operator template specialization has to be done. The corresponding
specialization is that instantiation of the function call operator template with the same template arguments as those
deduced for the conversion function template. Consider the following:

    auto glambda = []<auto a> { return a; };
int (*fp)(int) = glambda;

The behavior of the conversion function of glambda above is like that of the following conversion function:

    struct Closure
        { template<class T> auto operator()(T t) const { /* ... */ } }
        template<class T> static auto lambda_call_operator_invoker(T a) {
            // forwards execution to operator() (a) and therefore has
            // the same return type deduced

§ 7.5.5.2 106
template<class T> using fptr_t = 
decltype(lambda_call_operator_invoker(declval<T>()))(T);

template<class T> operator fptr_t<T>() const
{ return &lambda_call_operator_invoker; }

Example 5:

```cpp
void f1(int (*)(int)) { }
void f2(char (*)(int)) { }

void g(int (*)(int)) { } // #1
void g(char (*)(char)) { } // #2

void h(int (*)(int)) { } // #3
void h(char (*)(int)) { } // #4

auto glambda = []{auto a) { return a; };
f1(glmabda); // OK
f2(glmabda); // error: ID is not convertible
g(glmabda); // error: ambiguous
h(glmabda); // OK: calls #3 since it is convertible from ID
int& (*fpi)(int*) = []{(auto* a) -> auto& { return *a; }}; // OK
```

Example 6:

```cpp
auto GL = []{auto a) { std::cout << a; return a; };
int (*GL_int)(int) = GL; // OK: through conversion function template
GL_int(3); // OK: same as GL(3)
```

Example 7:

```cpp
auto Fwd = []{int (*fp)(int), auto a) { return fp(a); };
auto C = []{auto a) { return a; };

static_assert(Fwd(C,3) == 3); // OK

// No specialization of the function call operator template can be constexpr (due to the local static).
auto NC = []{auto a) { static int s; return a; };
static_assert(Fwd(NC,3) == 3); // error
```

The lambda-expression’s compound-statement yields the function-body (9.5) of the function call operator, but for purposes of name lookup (6.5), determining the type and value of this (11.4.3.2) and transforming id-expressions referring to non-static class members into class member access expressions using (*this) (11.4.3), the compound-statement is considered in the context of the lambda-expression.
struct S1 {
    int x, y;
    int operator()(int);
    void f() {
        [=]() -> int {
            return operator()(this->x + y);
            // equivalent to S1::operator()(this->x + (*this).y)
            // this has type S1*
        };
    }
};

—end example—

Further, a variable __func__ is implicitly defined at the beginning of the compound-statement of the lambda-expression, with semantics as described in 9.5.1.

13 The closure type associated with a lambda-expression has no default constructor if the lambda-expression has a lambda-capture and a defaulted default constructor otherwise. It has a defaulted copy constructor and a defaulted move constructor (11.4.5.3). It has a deleted copy assignment operator if the lambda-expression has a lambda-capture and defaulted copy and move assignment operators otherwise (11.4.6).

[Note 7: These special member functions are implicitly defined as usual, and might therefore be defined as deleted. —end note—]

14 The closure type associated with a lambda-expression has an implicitly-declared destructor (11.4.7).

15 A member of a closure type shall not be explicitly instantiated (13.9.3), explicitly specialized (13.9.4), or named in a friend declaration (11.9.4).

7.5.5.3 Captures

lambda-capture:

    capture-default
    capture-list
    capture-default , capture-list

capture-default:

    &
    =

capture-list:

    capture
    capture-list , capture

capture:

    simple-capture
    init-capture

simple-capture:

    identifier . . . opt
    & identifier . . . opt
    this
    * this

init-capture:

    . . . opt identifier initializer
    & . . . opt identifier initializer

1 The body of a lambda-expression may refer to local entities of enclosing block scopes by capturing those entities, as described below.

2 If a lambda-capture includes a capture-default that is &, no identifier in a simple-capture of that lambda-capture shall be preceded by &. If a lambda-capture includes a capture-default that is =, each simple-capture of that lambda-capture shall be of the form “& identifier . . . opt”, “this”, or “* this”.

[Note 1: The form [&,this] is redundant but accepted for compatibility with ISO C++ 2014. —end note—]

Ignoring appearances in initializers of init-captures, an identifier or this shall not appear more than once in a lambda-capture.

[Example 1:

    struct S2 { void f(int i); void f() { /* code */ }];

§ 7.5.5.3

108
void S2::f(int i) {
    [&, i](); // OK
    [&, this, i](); // OK, equivalent to [&, i]
    [&, &i](); // error: i preceded by & when & is the default
    [=, *this](); // OK
    [=, this](); // OK, equivalent to [=]
    [i, i](); // error: i repeated
    [this, *this](); // error: this appears twice
}

—end example

3 A lambda-expression shall not have a capture-default or simple-capture in its lambda-introducer unless its innermost enclosing scope is a block scope (6.4.3) or it appears within a default member initializer and its innermost enclosing scope is the corresponding class scope (6.4.7).

4 The identifier in a simple-capture is looked up using the usual rules for unqualified name lookup (6.5.2); each such lookup shall find a local entity. The simple-captures this and *this denote the local entity *this. An entity that is designated by a simple-capture is said to be explicitly captured.

5 If an identifier in a simple-capture appears as the declarator-id of a parameter of the lambda-declarator’s parameter-declaration-clause, the program is ill-formed.

[Example 2:

    void f() {
        int x = 0;
        auto g = [x](int x) { return 0; }; // error: parameter and simple-capture have the same name
    }

—end example]

6 An init-capture without ellipsis behaves as if it declares and explicitly captures a variable of the form “auto init-capture ;” whose declarative region is the lambda-expression’s compound-statement, except that:

(6.1) — if the capture is by copy (see below), the non-static data member declared for the capture and the variable are treated as two different ways of referring to the same object, which has the lifetime of the non-static data member, and no additional copy and destruction is performed, and

(6.2) — if the capture is by reference, the variable’s lifetime ends when the closure object’s lifetime ends.

[Note 2: This enables an init-capture like “x = std::move(x)”; the second “x” must bind to a declaration in the surrounding context. — end note]

[Example 3:

    int x = 4;
    auto y = [&r = x, x = x+1]()->int {
        r += 2;
        return x+2;
    }(); // Updates ::x to 6, and initializes y to 7.
    auto z = [a = 42](int a) { return 1; }; // error: parameter and local variable have the same name

—end example]

7 For the purposes of lambda capture, an expression potentially references local entities as follows:

(7.1) — An id-expression that names a local entity potentially references that entity; an id-expression that names one or more non-static class members and does not form a pointer to member (7.6.2.2) potentially references *this.

[Note 3: This occurs even if overload resolution selects a static member function for the id-expression. — end note]

(7.2) — A this expression potentially references *this.

(7.3) — A lambda-expression potentially references the local entities named by its simple-captures.

If an expression potentially references a local entity within a declarative region in which it is odr-usable, and the expression would be potentially evaluated if the effect of any enclosing typeid expressions (7.6.1.8) were ignored, the entity is said to be implicitly captured by each intervening lambda-expression with an associated capture-default that does not explicitly capture it. The implicit capture of *this is deprecated when the capture-default is =; see D.3.
Example 4:

```c
void f(int, const int (&)[2] = {});  // #1
void f(const int&, const int (&)[1]);  // #2

void test() {
    const int x = 17;
    auto g = [](auto a) {
        f(x);  // OK: calls #1, does not capture x
    };

    auto g1 = [=](auto a) {
        f(x);  // OK: calls #1, captures x
    };

    auto g2 = [=](auto a) {
        int selector[sizeof(a) == 1 ? 1 : 2]{};
        f(x, selector);  // OK: captures x, might call #1 or #2
    };

    auto g3 = [=](auto a) {
        typeid(a + x);  // captures x regardless of whether a + x is an unevaluated operand
    };
}
```

Within `g1`, an implementation might optimize away the capture of `x` as it is not odr-used.

---

Note 4: The set of captured entities is determined syntactically, and entities might be implicitly captured even if the expression denoting a local entity is within a discarded statement (8.5.2).

Example 5:

```c
template<bool B>
void f(int n) {
    [=](auto a) {
        if constexpr (B && sizeof(a) > 4) {
            (void)n;  // captures n regardless of the value of B and sizeof(int)
        }
    }(0);
}
```

---

Note 5: As a consequence, if a lambda-expression explicitly captures an entity that is not odr-usable, the program is ill-formed. (6.3).

---

Example 6:

```c
void f1(int i) {
    int const N = 20;
    auto m1 = [=]{
        int const M = 30;
        auto m2 = [=]{
            int x[N][M];  // OK: N and M are not odr-used
            x[0][0] = i;  // OK: i is explicitly captured by m2 and implicitly captured by m1
        };
    };

    struct s1 {
        int f;
    }

    void work(int n) {
        int m = n*n;
        int j = 40;
        auto m3 = [this,m] {
            auto m4 = [&,j] {
                int x = n;
                x += m;  // OK: m implicitly captured by m4 and explicitly captured by m3
            };
        };
    }
```

---

An entity is captured if it is captured explicitly or implicitly. An entity captured by a lambda-expression is odr-used in the scope containing the lambda-expression.

Note 5: As a consequence, if a lambda-expression explicitly captures an entity that is not odr-usable, the program is ill-formed. (6.3).

---

§ 7.5.5.3
\[
x += i;  \quad \text{// error: } i \text{ is odr-used but not odr-usable}
\]
\[
x += f;  \quad \text{// due to intervening function and class scopes}
\]
\[
\]
void f(const int*);
void g() {
    const int N = 10;
    [=] {
        int arr[N];     // OK: not an odr-use, refers to automatic variable
        f(&N);           // OK: causes N to be captured; &N points to
                          // the corresponding member of the closure type
    }
}
—end example

An entity is captured by reference if it is implicitly or explicitly captured but not captured by copy. It is unspecified whether additional unnamed non-static data members are declared in the closure type for entities captured by reference. If declared, such non-static data members shall be of literal type.

[Example 9:

// The inner closure type must be a literal type regardless of how reference captures are represented.
static_assert([](int n) { return [&n] { return ++n; }(); })(3) == 4);
—end example

A bit-field or a member of an anonymous union shall not be captured by reference.

An id-expression within the compound-statement of a lambda-expression that is an odr-use of a reference captured by reference refers to the entity to which the captured reference is bound and not to the captured reference.

[Note 8: The validity of such captures is determined by the lifetime of the object to which the reference refers, not by the lifetime of the reference itself. —end note]

[Example 10:

auto h(int &r) {
    return [&] {
        ++r;     // Valid after h returns if the lifetime of the
                  // object to which r is bound has not ended
    }
}
—end example

If a lambda-expression m2 captures an entity and that entity is captured by an immediately enclosing lambda-expression m1, then m2’s capture is transformed as follows:

(14.1) — if m1 captures the entity by copy, m2 captures the corresponding non-static data member of m1’s closure type;

(14.2) — if m1 captures the entity by reference, m2 captures the same entity captured by m1.

[Example 11: The nested lambda-expressions and invocations below will output 123234.

```cpp
int a = 1, b = 1, c = 1;
auto m1 = [a, &b, &c]() mutable {
    auto m2 = [a, b, &c]() mutable {
        std::cout << a << b << c;
        a = 4; b = 4; c = 4;
    }
    a = 3; b = 3; c = 3;
    m2();
};
a = 2; b = 2; c = 2;
m1();
std::cout << a << b << c;
—end example
```

When the lambda-expression is evaluated, the entities that are captured by copy are used to direct-initialize each corresponding non-static data member of the resulting closure object, and the non-static data members corresponding to the init-captures are initialized as indicated by the corresponding initializer (which may be copy- or direct-initialization). (For array members, the array elements are direct-initialized in increasing
subscript order.) These initializations are performed in the (unspecified) order in which the non-static data members are declared.

[Note 9: This ensures that the destructions will occur in the reverse order of the constructions. —end note]

[Note 10: If a non-reference entity is implicitly or explicitly captured by reference, invoking the function call operator of the corresponding lambda-expression after the lifetime of the entity has ended is likely to result in undefined behavior. —end note]

A simple-capture containing an ellipsis is a pack expansion (13.7.4). An init-capture containing an ellipsis is a pack expansion that introduces an init-capture pack (13.7.4) whose declarative region is the lambda-expression’s compound-statement.

[Example 12:]

```cpp
template<class... Args>
void f(Args... args) {
    auto lm = [& , args ...] { return g(args ...); };
    lm();

    auto lm2 = [...xs=std::move(args)] { return g(xs ...); };
    lm2();
}
—end example]
```

7.5.6 Fold expressions

A fold expression performs a fold of a pack (13.7.4) over a binary operator.

fold-expression:

```cpp
    ( cast-expression fold-operator ... )
    ( ... fold-operator cast-expression )
    ( cast-expression fold-operator ... fold-operator cast-expression )
```

fold-operator: one of

```
    + - * / % ^ & | << >>
    += -= *= /= %= ^= &= |= <<= >>= =
```

An expression of the form (... op e) where op is a fold-operator is called a unary left fold. An expression of the form (e op ...) where op is a fold-operator is called a unary right fold. Unary left folds and unary right folds are collectively called unary folds. In a unary fold, the cast-expression shall contain an unexpanded pack (13.7.4).

An expression of the form (e1 op1 ... op2 e2) where op1 and op2 are fold-operators is called a binary fold. In a binary fold, op1 and op2 shall be the same fold-operator, and either e1 shall contain an unexpanded pack or e2 shall contain an unexpanded pack, but not both. If e2 contains an unexpanded pack, the expression is called a binary left fold. If e1 contains an unexpanded pack, the expression is called a binary right fold.

[Example 1:]

```cpp
template<typename ...Args>
bool f(Args ...args) {
    return (true && ... && args); // OK
}

template<typename ...Args>
bool f(Args ...args) {
    return (args + ... + args); // error: both operands contain unexpanded packs
}
—end example]
```

7.5.7 Requires expressions

7.5.7.1 General

A requires-expression provides a concise way to express requirements on template arguments that can be checked by name lookup (6.5) or by checking properties of types and expressions.

requires-expression:

```
    requires requirement-parameter-list op requirement-body
```
requirement-parameter-list:
  { parameter-declaration-clause_opt }

requirement-body:
  { requirement-seq }

requirement-seq:
  requirement
  requirement-seq requirement

requirement:
  simple-requirement
  type-requirement
  compound-requirement
  nested-requirement

2 A requires-expression is a prvalue of type bool whose value is described below. Expressions appearing within a requirement-body are unevaluated operands (7.2).

3 [Example 1]: A common use of requires-expressions is to define requirements in concepts such as the one below:

```cpp
template<typename T>
concept R = requires (T i) {
    typename T::type;
    {*i} -> std::convertible_to<const typename T::type&>;
};
```

A requires-expression can also be used in a requires-clause (13.1) as a way of writing ad hoc constraints on template arguments such as the one below:

```cpp
template<typename T>
requires requires (T x) { x + x; }
T add(T a, T b) { return a + b; }
```

The first requires introduces the requires-clause, and the second introduces the requires-expression. —end example]

4 A requires-expression may introduce local parameters using a parameter-declaration-clause (9.3.4.6). A local parameter of a requires-expression shall not have a default argument. Each name introduced by a local parameter is in scope from the point of its declaration until the closing brace of the requirement-body. These parameters have no linkage, storage, or lifetime; they are only used as notation for the purpose of defining requirements. The parameter-declaration-clause of a requirement-parameter-list shall not terminate with an ellipsis.

[Example 2]:

```cpp
template<typename T>
concept C = requires(T t, ...) {
    new int[-(int)sizeof(T)];  // ill-formed, no diagnostic required
};
```

—end example]

5 The requirement-body contains a sequence of requirements. These requirements may refer to local parameters, template parameters, and any other declarations visible from the enclosing context.

6 The substitution of template arguments into a requires-expression may result in the formation of invalid types or expressions in its requirements or the violation of the semantic constraints of those requirements. In such cases, the requires-expression evaluates to false; it does not cause the program to be ill-formed. The substitution and semantic constraint checking proceeds in lexical order and stops when a condition that determines the result of the requires-expression is encountered. If substitution (if any) and semantic constraint checking succeed, the requires-expression evaluates to true.

[Note 1]: If a requires-expression contains invalid types or expressions in its requirements, and it does not appear within the declaration of a templated entity, then the program is ill-formed. — end note]

If the substitution of template arguments into a requirement would always result in a substitution failure, the program is ill-formed; no diagnostic required.

[Example 3]:

```cpp
template<typename T> concept C =
  requires {
    new int[-(int)sizeof(T)];  // ill-formed, no diagnostic required
  };
```

§ 7.5.7.1 114
7.5.7.2 Simple requirements

Simple-requirement:

expression ;

A simple-requirement asserts the validity of an expression.

[Note 1: The enclosing requires-expression will evaluate to false if substitution of template arguments into the expression fails. The expression is an unevaluated operand (7.2). — end note]

[Example 1:]

```cpp
template<typename T> concept C =
    requires (T a, T b) {
        a + b; // C<T> is true if a + b is a valid expression
    };
```

— end example]

2 A requirement that starts with a requires token is never interpreted as a simple-requirement.

[Note 2: This simplifies distinguishing between a simple-requirement and a nested-requirement. — end note]

7.5.7.3 Type requirements

type-requirement:

type-name ;

A type-requirement asserts the validity of a type.

[Note 1: The enclosing requires-expression will evaluate to false if substitution of template arguments fails. — end note]

[Example 1:]

```cpp
template<typename T, typename T::type = 0> struct S;
template<typename T> using Ref = T&;
template<typename T> concept C = requires {
    typename T::inner; // required nested member name
    typename S<T>; // required class template specialization
    typename Ref<T>; // required alias template substitution, fails if T is void
};
```

— end example]

2 A type-requirement that names a class template specialization does not require that type to be complete (6.8).

7.5.7.4 Compound requirements

Compound-requirement:

{ expression } noexcept opt return-type-requirement opt ;

return-type-requirement:

-> type-constraint

A compound-requirement asserts properties of the expression E. Substitution of template arguments (if any) and verification of semantic properties proceed in the following order:

1. Substitution of template arguments (if any) into the expression is performed.
2. If the noexcept specifier is present, E shall not be a potentially-throwing expression (14.5).
3. If the return-type-requirement is present, then:
   1. Substitution of template arguments (if any) into the return-type-requirement is performed.
   2. The immediately-declared constraint (13.2) of the type-constraint for decltype((E)) shall be satisfied.

[Example 1: Given concepts C and D,

```cpp
requires {
    { E1 } -> C;
    { E2 } -> D<A1, ..., An>;
};
```
is equivalent to

```c++
requires {
    E1; requires C<decltype((E1))>;
    E2; requires D<decltype((E2)), A_1, ···, A_n>;
};
```

(including in the case where n is zero). — end example

Example 2:

```c++
template<typename T> concept C1 = requires(T x) {
    {x++};
};
```

The compound-requirement in C1 requires that `x++` is a valid expression. It is equivalent to the simple-requirement `x++;`.

```c++
template<typename T> concept C2 = requires(T x) {
    {*x} -> std::same_as<typename T::inner>;
};
```

The compound-requirement in C2 requires that `*x` is a valid expression, that `typename T::inner` is a valid type, and that `std::same_as<decltype(*x), typename T::inner>` is satisfied.

```c++
template<typename T> concept C3 = requires(T x) {
    {g(x)} noexcept;
};
```

The compound-requirement in C3 requires that `g(x)` is a valid expression and that `g(x)` is non-throwing. — end example

7.5.7.5 Nested requirements

```c++
nested-requirement:
    requires constraint-expression ;
```

A nested-requirement can be used to specify additional constraints in terms of local parameters. The constraint-expression shall be satisfied (13.5.3) by the substituted template arguments, if any. Substitution of template arguments into a nested-requirement does not result in substitution into the constraint-expression other than as specified in 13.5.2.

Example 1:

```c++
template<typename U> concept C = sizeof(U) == 1;

template<typename T> concept D = requires (T t) {
    requires C<decltype (+t)>
};
```

D<T> is satisfied if `sizeof(decltype (+t)) == 1` (13.5.2.3). — end example

Example 2:

```c++
template<typename T> concept C = requires (T a) {
    requires sizeof(a) == 4; // OK
    requires a == 0; // error: evaluation of a constraint variable
};
```

A local parameter shall only appear as an unevaluated operand (7.2) within the constraint-expression.

Example 2:

```c++
template<typename T> concept C = requires (T a) {
    requires sizeof(a) == 4; // OK
    requires a == 0; // error: evaluation of a constraint variable
};
```

— end example

7.6 Compound expressions

7.6.1 Postfix expressions

7.6.1.1 General
postfix-expression:
  primary-expression
  postfix-expression [ expr-or-braced-init-list ]
  postfix-expression ( expression-list_opt )
  simple-type-specifier ( expression-list_opt )
  typename-specifier ( expression-list_opt )
  simple-type-specifier braced-init-list
  typename-specifier braced-init-list
  postfix-expression . template_opt id-expression
  postfix-expression -> template_opt id-expression
  postfix-expression ++
  postfix-expression --
  dynamic_cast < type-id > ( expression )
  static_cast < type-id > ( expression )
  reinterpret_cast < type-id > ( expression )
  const_cast < type-id > ( expression )
  typeid ( expression )
  typeid ( type-id )

expression-list:
  initializer-list

2 [Note 1: The > token following the type-id in a dynamic_cast, static_cast, reinterpret_cast, or const_cast might be the product of replacing a >> token by two consecutive > tokens (13.3). —end note]

7.6.1.2 Subscripting [expr.sub]

1 A postfix expression followed by an expression in square brackets is a postfix expression. One of the expressions shall be a glvalue of type “array of T” or a prvalue of type “pointer to T” and the other shall be a prvalue of unscoped enumeration or integral type. The result is of type “T”. The type “T” shall be a completely-defined object type. The expression \(E_1[E_2]\) is identical (by definition) to \(*((E_1)+(E_2))\), except that in the case of an array operand, the result is an lvalue if that operand is an lvalue and an xvalue otherwise. The expression \(E_1\) is sequenced before the expression \(E_2\).

2 [Note 1: A comma expression (7.6.20) appearing as the expr-or-braced-init-list of a subscripting expression is deprecated; see D.4. —end note]

3 [Note 2: Despite its asymmetric appearance, subscripting is a commutative operation except for sequencing. See 7.6.2 and 7.6.6 for details of * and + and 9.3.4.5 for details of array types. —end note]

4 A braced-init-list shall not be used with the built-in subscript operator.

7.6.1.3 Function call [expr.call]

1 A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of initializer-clauses which constitute the arguments to the function.

[Note 1: If the postfix expression is a function or member function name, the appropriate function and the validity of the call are determined according to the rules in 12.4. —end note]

The postfix expression shall have function type or function pointer type. For a call to a non-member function or to a static member function, the postfix expression shall either be an lvalue that refers to a function (in which case the function-to-pointer standard conversion (7.3.4) is suppressed on the postfix expression), or have function pointer type.

2 For a call to a non-static member function, the postfix expression shall be an implicit (11.4.3, 11.4.9) or explicit class member access (7.6.1.5) whose id-expression is a function member name, or a pointer-to-member expression (7.6.4) selecting a function member; the call is as a member of the class object referred to by the object expression. In the case of an implicit class member access, the implied object is the one pointed to by this.

[Note 2: A member function call of the form \(f()\) is interpreted as \(*\text{this}).f()\) (see 11.4.3). —end note]

3 If the selected function is non-virtual, or if the id-expression in the class member access expression is a qualified-id, that function is called. Otherwise, its final overrider (11.7.3) in the dynamic type of the object expression is called; such a call is referred to as a virtual function call.

62) This is true even if the subscript operator is used in the following common idiom: \&x[0].
[Note 3: The dynamic type is the type of the object referred to by the current value of the object expression. 11.10.5 describes the behavior of virtual function calls when the object expression refers to an object under construction or destruction. — end note]

4 [Note 4: If a function or member function name is used, and name lookup (6.5) does not find a declaration of that name, the program is ill-formed. No function is implicitly declared by such a call. — end note]

5 If the postfix-expression names a destructor or pseudo-destructor (7.5.4.4), the type of the function call expression is void; otherwise, the type of the function call expression is the return type of the statically chosen function (i.e., ignoring the virtual keyword), even if the type of the function actually called is different. This return type shall be an object type, a reference type or cv void. If the postfix-expression names a pseudo-destructor (in which case the postfix-expression is a possibly-parenthesized class member access), the function call destroys the object of scalar type denoted by the object expression of the class member access (7.6.1.5, 6.7.3).

6 Calling a function through an expression whose function type is different from the function type of the called function’s definition results in undefined behavior.

7 When a function is called, each parameter (9.3.4.6) is initialized (9.4, 11.4.5.3) with its corresponding argument. If there is no corresponding argument, the default argument for the parameter is used.

[Example 1:]
```cpp
    template<typename ...T> int f(int n = 0, T ...t);
    int x = f<int>();           // error: no argument for second function parameter
   — end example]
```

If the function is a non-static member function, the this parameter of the function (11.4.3.2) is initialized with a pointer to the object of the call, converted as if by an explicit type conversion (7.6.3).

[Note 5: There is no access or ambiguity checking on this conversion; the access checking and disambiguation are done as part of the (possibly implicit) class member access operator. See 11.8, 11.9.3, and 7.6.1.5. — end note]

When a function is called, the type of any parameter shall not be a class type that is either incomplete or abstract.

[Note 6: This still allows a parameter to be a pointer or reference to such a type. However, it prevents a passed-by-value parameter to have an incomplete or abstract class type. — end note]

It is implementation-defined whether the lifetime of a parameter ends when the function in which it is defined returns or at the end of the enclosing full-expression. The initialization and destruction of each parameter occurs within the context of the calling function.

[Example 2: The access of the constructor, conversion functions or destructor is checked at the point of call in the calling function. If a constructor or destructor for a function parameter throws an exception, the search for a handler starts in the scope of the calling function; in particular, if the function called has a function-try-block (14.1) with a handler that could handle the exception, this handler is not considered. — end example]

8 The postfix-expression is sequenced before each expression in the expression-list and any default argument. The initialization of a parameter, including every associated value computation and side effect, is indeterminately sequenced with respect to that of any other parameter.

[Note 7: All side effects of argument evaluations are sequenced before the function is entered (see 6.9.1). — end note]

[Example 3:]
```cpp
    void f() {
        std::string s = "but I have heard it works even if you don’t believe in it";
        s.replace(0, 4, "").replace(s.find("even"), 4, "only").replace(s.find(" don’t"), 6, "");
        assert(s == "I have heard it works only if you believe in it");       // OK
    }
   — end example]
```

[Note 8: If an operator function is invoked using operator notation, argument evaluation is sequenced as specified for the built-in operator; see 12.4.2.3. — end note]

[Example 4:]
```cpp
    struct S {
        S(int);
    };
    int operator<(S, int);
    int i, j;
```


```c
int x = S(i=1) << (i=2);
int y = operator<<(S(j=1, j=2);
```

After performing the initializations, the value of i is 2 (see 7.6.7), but it is unspecified whether the value of j is 1 or 2. — end example

The result of a function call is the result of the possibly-converted operand of the return statement (8.7.4) that transferred control out of the called function (if any), except in a virtual function call if the return type of the final overrider is different from the return type of the statically chosen function, the value returned from the final overrider is converted to the return type of the statically chosen function.

[Note 9: A function can change the values of its non-const parameters, but these changes cannot affect the values of the arguments except where a parameter is of a reference type (9.3.4.3); if the reference is to a const-qualified type, const_cast is required to be used to cast away the constness in order to modify the argument’s value. Where a parameter is of const reference type a temporary object is introduced if needed (9.2.9, 5.13, 5.13.5, 9.3.4.5, 6.7.7). In addition, it is possible to modify the values of non-constant objects through pointer parameters. — end note]

A function can be declared to accept fewer arguments (by declaring default arguments (9.3.4.7)) or more arguments (by using the ellipsis, ..., or a function parameter pack (9.3.4.6)) than the number of parameters in the function definition (9.5).

[Note 10: This implies that, except where the ellipsis (...) or a function parameter pack is used, a parameter is available for each argument. — end note]

When there is no parameter for a given argument, the argument is passed in such a way that the receiving function can obtain the value of the argument by invoking va_arg (17.13).

[Note 11: This paragraph does not apply to arguments passed to a function parameter pack. Function parameter packs are expanded during template instantiation (13.7.4), thus each such argument has a corresponding parameter when a function template specialization is actually called. — end note]

The lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) standard conversions are performed on the argument expression. An argument that has type cv std::nullptr_t is converted to type void* (7.3.12). After these conversions, if the argument does not have arithmetic, enumeration, pointer, pointer-to-member, or class type, the program is ill-formed. Passing a potentially-evaluated argument of a scoped enumeration type or of a class type (Clause 11) having an eligible non-trivial copy constructor, an eligible non-trivial move constructor, or a non-trivial destructor (11.4.4), with no corresponding parameter, is conditionally-supported with implementation-defined semantics. If the argument has integral or enumeration type that is subject to the integral promotions (7.3.7), or a floating-point type that is subject to the floating-point promotion (7.3.8), the value of the argument is converted to the promoted type before the call. These promotions are referred to as the default argument promotions.

Recursive calls are permitted, except to the main function (6.9.3.1).

A function call is an lvalue if the result type is an lvalue reference type or an rvalue reference to function type, an xvalue if the result type is an rvalue reference to object type, and a prvalue otherwise.

### 7.6.1.4 Explicit type conversion (functional notation) [expr.type.conv]

1. A simple-type-specifier (9.2.9.3) or typename-specifier (13.8) followed by a parenthesized optional expression-list or by a braced-init-list (the initializer) constructs a value of the specified type given the initializer. If the type is a placeholder for a deduced class type, it is replaced by the return type of the function selected by overload resolution for class template deduction (12.4.2.9) for the remainder of this subclause.

2. If the initializer is a parenthesized single expression, the type conversion expression is equivalent to the corresponding cast expression (7.6.3). Otherwise, if the type is cv void and the initializer is () or {} (after pack expansion, if any), the expression is a prvalue of the specified type that performs no initialization. Otherwise, the expression is a prvalue of the specified type whose result object is direct-initialized (9.4) with the initializer. If the initializer is a parenthesized optional expression-list, the specified type shall not be an array type.

### 7.6.1.5 Class member access [expr.ref]

1. A postfix expression followed by a dot . or an arrow ->, optionally followed by the keyword template (13.3), and then followed by an id-expression, is a postfix expression. The postfix expression before the dot or arrow is evaluated, the result of that evaluation, together with the id-expression, determines the result of the entire postfix expression.

---

© ISO/IEC N4868

### § 7.6.1.5

---

(63) If the class member access expression is evaluated, the subexpression evaluation happens even if the result is unnecessary to determine the value of the entire postfix expression, for example if the id-expression denotes a static member.
2 For the first option (dot) the first expression shall be a glvalue. For the second option (arrow) the first expression shall be a prvalue having pointer type. The expression E1->E2 is converted to the equivalent form (*((E1)).E2; the remainder of 7.6.1.5 will address only the first option (dot).\footnote{Note that (*((E1))) is an lvalue.}

3 Abbreviating \textit{postfix-expression.id-expression} as E1.E2, E1 is called the \textit{object expression}. If the object expression is of scalar type, E2 shall name the pseudo-destructor of that same type (ignoring cv-qualifications) and E1.E2 is an lvalue of type \textit{“function of () returning void”}. 

\textit{[Note 1: This value can only be used for a notional function call (7.5.4.4). — end note]}

4 Otherwise, the object expression shall be of class type. The class type shall be complete unless the class member access appears in the definition of that class.

\textit{[Note 2: If the class is incomplete, lookup in the complete class type is required to refer to the same declaration (6.4.7). — end note]}

The \textit{id-expression} shall name a member of the class or of one of its base classes.

\textit{[Note 3: Because the name of a class is inserted in its class scope (Clause 11), the name of a class is also considered a nested member of that class. — end note]}

\textit{[Note 4: 6.5.6 describes how names are looked up after the . and \textit{->} operators. — end note]}

5 If E2 is a bit-field, E1.E2 is a bit-field. The type and value category of E1.E2 are determined as follows. In the remainder of 7.6.1.5, cq represents either \textit{const} or the absence of \textit{const} and vq represents either \textit{volatile} or the absence of \textit{volatile}. cv represents an arbitrary set of cv-qualifiers, as defined in 6.8.4.

6 If E2 is declared to have type \textit{“reference to T”}, then E1.E2 is an lvalue; the type of E1.E2 is T. Otherwise, one of the following rules applies.

\begin{enumerate}
\item If E2 is a static data member and the type of E2 is T, then E1.E2 is an lvalue; the expression designates the named member of the class. The type of E1.E2 is T.
\item If E2 is a non-static data member and the type of E1 is \textit{“cq1 vq1 X”}, and the type of E2 is \textit{“cq2 vq2 T”}, the expression designates the corresponding member subobject of the object designated by the first expression. If E1 is an lvalue, then E1.E2 is an lvalue; otherwise E1.E2 is an rvalue. Let the notation vq12 stand for the “union” of vq1 and vq2; that is, if vq1 or vq2 is \textit{volatile}, then vq12 is \textit{volatile}. Similarly, let the notation cq12 stand for the “union” of cq1 and cq2; that is, if cq1 or cq2 is \textit{const}, then cq12 is \textit{const}. If E2 is declared to be a \textit{mutable} member, then the type of E1.E2 is \textit{“vq12 T”}. If E2 is not declared to be a \textit{mutable} member, then the type of E1.E2 is \textit{“cq12 vq12 T”}.
\item If E2 is a (possibly overloaded) member function, function overload resolution (12.4) is used to select the function to which E2 refers. The type of E1.E2 is the type of E2 and E1.E2 refers to the function referred to by E2.
\begin{enumerate}
\item If E2 refers to a static member function, E1.E2 is an lvalue.
\item Otherwise (when E2 refers to a non-static member function), E1.E2 is a prvalue. The expression can be used only as the left-hand operand of a member function call (11.4.2).
\textit{[Note 5: Any redundant set of parentheses surrounding the expression is ignored (7.5.3). — end note]}
\end{enumerate}
\item If E2 is a nested type, the expression E1.E2 is ill-formed.
\item If E2 is a member enumerator and the type of E2 is T, the expression E1.E2 is a prvalue. The type of E1.E2 is T.
\end{enumerate}

7 If E2 is a non-static data member or a non-static member function, the program is ill-formed if the class of which E2 is directly a member is an ambiguous base (11.8) of the naming class (11.9.3) of E2.

\textit{[Note 6: The program is also ill-formed if the naming class is an ambiguous base of the class type of the object expression; see 11.9.3. — end note]}

7.6.1.6 \textit{Increment and decrement} \footnote{Expression} [expr.post.incr]

1 The value of a postfix ++ expression is the value of its operand.

\textit{[Note 1: The value obtained is a copy of the original value. — end note]}

The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type other than \textit{cv bool}, or a pointer to a complete object type. An operand with volatile-qualified type is deprecated; see D.6. The value of the operand object is modified (3.1) by adding 1 to it. The value computation of the ++ expression
is sequenced before the modification of the operand object. With respect to an indeterminately-sequenced function call, the operation of postfix ++ is a single evaluation.

[Note 2: Therefore, a function call cannot intervene between the lvalue-to-rvalue conversion and the side effect associated with any single postfix ++ operator. — end note]

The result is a prvalue. The type of the result is the cv-unqualified version of the type of the operand. If the operand is a bit-field that cannot represent the incremented value, the resulting value of the bit-field is implementation-defined. See also 7.6.6 and 7.6.19.

The operand of postfix -- is decremented analogously to the postfix ++ operator.

[Note 3: For prefix increment and decrement, see 7.6.2.3. — end note]

7.6.1.7 Dynamic cast [expr.dynamic.cast]

The result of the expression $\text{dynamic\_cast}<T>(v)$ is the result of converting the expression $v$ to type $T$. $T$ shall be a pointer or reference to a complete class type, or “pointer to $\text{cv void}$". The $\text{dynamic\_cast}$ operator shall not cast away constness (7.6.1.11).

2 If $T$ is a pointer type, $v$ shall be a prvalue of a pointer to complete class type, and the result is a prvalue of type $T$. If $T$ is an lvalue reference type, $v$ shall be an lvalue of a complete class type, and the result is an lvalue of the type referred to by $T$. If $T$ is an rvalue reference type, $v$ shall be a glvalue having a complete class type, and the result is an xvalue of the type referred to by $T$.

3 If the type of $v$ is the same as $T$ (ignoring cv-qualifications), the result is $v$ (converted if necessary).

4 If $T$ is “pointer to $\text{cv1 B}$” and $v$ has type “pointer to $\text{cv2 D}$” such that $B$ is a base class of $D$, the result is a pointer to the unique $B$ subobject of the $D$ object pointed to by $v$, or a null pointer value if $v$ is a null pointer value. Similarly, if $T$ is “reference to $\text{cv1 B}$” and $v$ has type $\text{cv2 D}$ such that $B$ is a base class of $D$, the result is the unique $B$ subobject of the $D$ object referred to by $v$. In both the pointer and reference cases, the program is ill-formed if $B$ is an inaccessible or ambiguous base class of $D$.

[Example 1:

```cpp
define B { }
define D : B { }
define void foo(D* dp) {
    B* bp = dynamic_cast<B*>(dp); // equivalent to B* bp = dp;
}
```

— end example]

5 Otherwise, $v$ shall be a pointer to or a glvalue of a polymorphic type (11.7.3).

6 If $v$ is a null pointer value, the result is a null pointer value.

7 If $T$ is “pointer to $\text{cv void}$”, then the result is a pointer to the most derived object pointed to by $v$. Otherwise, a runtime check is applied to see if the object pointed or referred to by $v$ can be converted to the type pointed or referred to by $T$.

8 If $C$ is the class type to which $T$ points or refers, the runtime check logically executes as follows:

(8.1) — If, in the most derived object pointed (referred) to by $v$, $v$ points (refers) to a public base class subobject of a $C$ object, and if only one object of type $C$ is derived from the subobject pointed (referred) to by $v$ the result points (refers) to that $C$ object.

(8.2) — Otherwise, if $v$ points (refers) to a public base class subobject of the most derived object, and the type of the most derived object has a base class, of type $C$, that is unambiguous and public, the result points (refers) to the $C$ subobject of the most derived object.

(8.3) — Otherwise, the runtime check fails.

9 The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws an exception (14.2) of a type that would match a handler (14.4) of type std::bad_cast (17.7.4).

[Example 2:

```cpp
class A { virtual void f(); }
class B { virtual void g(); }
class D : public virtual A, private B { }
```

65) The most derived object (6.7.2) pointed or referred to by $v$ can contain other $B$ objects as base classes, but these are ignored.
```c
void g() {
    D d;
    B* bp = (B*)&d; // cast needed to break protection
    A* ap = &d;   // public derivation, no cast needed
    D& dr = dynamic_cast<D&>(*bp); // fails
    ap = dynamic_cast<A*>(bp);    // fails
    bp = dynamic_cast<B*>(ap);    // ill-formed (not a runtime check)
    // succeeds
    ap = dynamic_cast<A*>(&d);
    bp = dynamic_cast<B*>(&d);
}

class E : public D, public B {
};
class F : public E, public D {
};
void h() {
    F f;
    A* ap = &f;  // succeeds: finds unique A
    D* dp = dynamic_cast<D*>(ap); // fails: yields null; f has two D subobjects
    E* ep = (E*)ap; // error: cast from virtual base
    E* ep1 = dynamic_cast<E*>(ap); // succeeds
}

—end example

[Note 1: Subclause 11.10.5 describes the behavior of a dynamic_cast applied to an object under construction or destruction. —end note]

7.6.1.8 Type identification

The result of a typeid expression is an lvalue of static type const std::type_info (17.7.3) and dynamic type const std::type_info or const name where name is an implementation-defined class publicly derived from std::type_info which preserves the behavior described in 17.7.3. The lifetime of the object referred to by the lvalue extends to the end of the program. Whether or not the destructor is called for the std::type_info object at the end of the program is unspecified.

When typeid is applied to a glvalue whose type is a polymorphic class type (11.7.3), the result refers to a std::type_info object representing the type of the most derived object (6.7.2) (that is, the dynamic type) to which the glvalue refers. If the glvalue is obtained by applying the unary * operator to a pointer and the pointer is a null pointer value (6.8.3), the typeid expression throws an exception (14.2) of a type that would match a handler of type std::bad_typeid exception (17.7.5).

When typeid is applied to an expression other than a glvalue of a polymorphic class type, the result refers to a std::type_info object representing the static type of the expression. Lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) conversions are not applied to the expression. If the expression is a prvalue, the temporary materialization conversion (7.3.5) is applied. The expression is an unevaluated operand (7.2).

When typeid is applied to a type-id, the result refers to a std::type_info object representing the type of the type-id. If the type of the type-id is a reference to a possibly cv-qualified type, the result of the typeid expression refers to a std::type_info object representing the cv-unqualified referenced type. If the type of the type-id is a class type or a reference to a class type, the class shall be completely-defined.

[Note 1: The type-id cannot denote a function type with a cv-qualifier-seq or a ref-qualifier (9.3.4.6). —end note]

If the type of the expression or type-id is a cv-qualified type, the result of the typeid expression refers to a std::type_info object representing the cv-unqualified type.

[Example 1:

class D { /* ... */ };  
D d1;  
const D d2;

typeid(d1) == typeid(D);     // yields true  
typeid(D) == typeid(const D); // yields true  
typeid(D) == typeid(D);      // yields true  
typeid(D) == typeid(const D&); // yields true

66) The recommended name for such a class is extended_type_info.
67) If p is an expression of pointer type, then *p, (*p), *(p), (**p), (**p), and so on all meet this requirement.

§ 7.6.1.8] 122
If the header `<typeinfo>` (17.7.3) is not imported or included prior to a use of `typeid`, the program is ill-formed.

[Note 2: Subclause 11.10.5 describes the behavior of `typeid` applied to an object under construction or destruction. —end note]

### 7.6.1.9 Static cast

The result of the expression `static_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue. The `static_cast` operator shall not cast away constness (7.6.1.11).

An lvalue of type “`cv1 B`”, where `B` is a class type, can be cast to type “reference to `cv2 D`”, where `D` is a class derived (11.7) from `B`, if `cv2` is the same cv-qualification as, or greater cv-qualification than, `cv1`. If `B` is a virtual base class of `D` or a base class of a virtual base class of `D`, or if no valid standard conversion from “pointer to `D`” to “pointer to `B`” exists (7.3.12), the program is ill-formed. An xvalue of type “`cv1 B`” can be cast to type “rvalue reference to `cv2 D`” with the same constraints as for an lvalue of type “`cv1 B`”. If the object of type “`cv1 B`” is actually a base class subobject of an object of type `D`, the result refers to the enclosing object of type `D`. Otherwise, the behavior is undefined.

#### Example 1:
```cpp
struct B {};
struct D : public B {};
D d;
B &br = d;
static_cast<D&>(br); // produces lvalue denoting the original d object
```

An lvalue of type `T1` can be cast to type “rvalue reference to `T2`” if `T2` is reference-compatible with `T1` (9.4.4). If the value is not a bit-field, the result refers to the object or the specified base class subobject thereof; otherwise, the lvalue-to-rvalue conversion (7.3.2) is applied to the bit-field and the resulting prvalue is used as the expression of the `static_cast` for the remainder of this subclause. If `T2` is an inaccessible (11.9) or ambiguous (11.8) base class of `T1`, a program that necessitates such a cast is ill-formed.

An expression `E` can be explicitly converted to a type `T` if there is an implicit conversion sequence (12.4.4.2) from `E` to `T`, if overload resolution for a direct-initialization (9.4) of an object or reference of type `T` from `E` would find at least one viable function (12.4.3), or if `T` is an aggregate type (9.4.2) having a first element `x` and there is an implicit conversion sequence from `E` to the type of `x`. If `T` is a reference type, the effect is the same as performing the declaration and initialization

\[
T t(E);
\]

for some invented temporary variable `t` (9.4) and then using the temporary variable as the result of the conversion. Otherwise, the result object is direct-initialized from `E`.

[Note 1: The conversion is ill-formed when attempting to convert an expression of class type to an inaccessible or ambiguous base class. —end note]

[Note 2: If `T` is “array of unknown bound of `U`”, this direct-initialization defines the type of the expression as `U[1]`. —end note]

Otherwise, the `static_cast` shall perform one of the conversions listed below. No other conversion shall be performed explicitly using a `static_cast`.

Any expression can be explicitly converted to type `cv void`, in which case it becomes a discarded-value expression (7.2).

[Note 3: However, if the value is in a temporary object (6.7.7), the destructor for that object is not executed until the usual time, and the value of the object is preserved for the purpose of executing the destructor. —end note]

The inverse of any standard conversion sequence (7.3) not containing an lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), function-to-pointer (7.3.4), null pointer (7.3.12), null member pointer (7.3.13), boolean (7.3.15), or function pointer (7.3.14) conversion, can be performed explicitly using `static_cast`. A program is ill-formed if it uses `static_cast` to perform the inverse of an ill-formed standard conversion sequence.

#### Example 2:

**§ 7.6.1.9**
The lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) conversions are applied to the operand. Such a \texttt{static\_cast} is subject to the restriction that the explicit conversion does not cast away constness (7.6.1.11), and the following additional rules for specific cases:

A value of a scoped enumeration type (9.7.1) can be explicitly converted to an integral type; the result is the same as that of converting to the enumeration’s underlying type and then to the destination type. A value of a scoped enumeration type can also be explicitly converted to a floating-point type; the result is the same as that of converting from the original value to the floating-point type.

A value of integral or enumeration type can be explicitly converted to a complete enumeration type. If the enumeration type has a fixed underlying type, the value is first converted to that type by integral conversion, if necessary, and then to the enumeration type. If the enumeration type does not have a fixed underlying type, the value is unchanged if the original value is within the range of the enumeration values (9.7.1), and otherwise, the behavior is undefined. A value of floating-point type can also be explicitly converted to an enumeration type. The resulting value is the same as converting the original value to the underlying type of the enumeration (7.3.11), and subsequently to the enumeration type.

A prvalue of type “pointer to \texttt{cv1} B”, where \texttt{B} is a class type, can be converted to a prvalue of type “pointer to \texttt{cv2} D”, where \texttt{D} is a complete class derived (11.7) from \texttt{B}, if \texttt{cv2} is the same \texttt{cv}-qualification as, or greater \texttt{cv}-qualification than, \texttt{cv1}. If \texttt{B} is a virtual base class of \texttt{D} or a base class of a virtual base class of \texttt{D}, or if no valid standard conversion from “pointer to \texttt{D}” to “pointer to \texttt{B}” exists (7.3.12), the program is ill-formed. The null pointer value (6.8.3) is converted to the null pointer value of the destination type. If the prvalue of type “pointer to \texttt{cv1} B” points to a \texttt{B} that is actually a subobject of an object of type \texttt{D}, the resulting pointer points to the enclosing object of type \texttt{D}. Otherwise, the behavior is undefined.

A prvalue of type “pointer to member of \texttt{D} of type \texttt{cv1} T” can be converted to a prvalue of type “pointer to member of \texttt{B} of type \texttt{cv2} T”, where \texttt{D} is a complete class type and \texttt{B} is a base class (11.7) of \texttt{D}, if \texttt{cv2} is the same \texttt{cv}-qualification as, or greater \texttt{cv}-qualification than, \texttt{cv1}.

If no valid standard conversion from “pointer to member of \texttt{B} of type \texttt{T}” to “pointer to member of \texttt{D} of type \texttt{T}” exists (7.3.13), the program is ill-formed. The null member pointer value (7.3.13) is converted to the null member pointer value of the destination type. If class \texttt{B} contains the original member, or is a base or derived class of the class containing the original member, the resulting pointer to member points to the original member. Otherwise, the behavior is undefined.

A prvalue of type “pointer to \texttt{cv1} void” can be converted to a prvalue of type “pointer to \texttt{cv2} T”, where \texttt{T} is an object type and \texttt{cv2} is the same \texttt{cv}-qualification as, or greater \texttt{cv}-qualification than, \texttt{cv1}. If the original pointer value represents the address \texttt{A} of a byte in memory and \texttt{A} does not satisfy the alignment requirement of \texttt{T}, then the resulting pointer value is unspecified. Otherwise, if the original pointer value points to an object \texttt{a}, and there is an object \texttt{b} of type \texttt{T} (ignoring \texttt{cv}-qualification) that is pointer-interconvertible (6.8.3) with \texttt{a}, the result is a pointer to \texttt{b}. Otherwise, the pointer value is unchanged by the conversion.

7.6.1.10 Reinterpret cast

The result of the expression \texttt{reinterpret\_cast\langle T\rangle(v)} is the result of converting the expression \texttt{v} to type \texttt{T}. If \texttt{T} is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if \texttt{T}
is an rvalue reference to object type, the result is an rvalue; otherwise, the result is a prvalue and the value to rvalue (7.3.2), array to pointer (7.3.3), and function to pointer (7.3.4) standard conversions are performed on the expression v. Conversions that can be performed explicitly using reinterpret_cast are listed below. No other conversion can be performed explicitly using reinterpret_cast.

2 The reinterpret_cast operator shall not cast away constness (7.6.1.11). An expression of integral, enumeration, pointer, or pointer to member type can be explicitly converted to its own type; such a cast yields the value of its operand.

   [Note 1: The mapping performed by reinterpret_cast might, or might not, produce a representation different from the original value. — end note]

3 A pointer can be explicitly converted to any integral type large enough to hold all values of its type. The mapping function is implementation-defined.

   [Note 2: It is intended to be unsurprising to those who know the addressing structure of the underlying machine. — end note]

A value of type std::nullptr_t can be converted to an integral type; the conversion has the same meaning and validity as a conversion of (void*)0 to the integral type.

   [Note 3: A reinterpret_cast cannot be used to convert a value of any type to the type std::nullptr_t. — end note]

4 A value of integral type or enumeration type can be explicitly converted to a pointer. A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined.

   [Note 4: Except as described in 6.7.5.5.4, the result of such a conversion will not be a safely-derived pointer value. — end note]

5 A function pointer can be explicitly converted to a function pointer of a different type.

   [Note 5: The effect of calling a function through a pointer to a function type (9.3.4.6) that is not the same as the type used in the definition of the function is undefined (7.6.1.3). — end note]

Except that converting a prvalue of type “pointer to T1” to the type “pointer to T2” (where T1 and T2 are function types) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.

   [Note 6: See also 7.3.12 for more details of pointer conversions. — end note]

6 An object pointer can be explicitly converted to an object pointer of a different type. When a prvalue v of object pointer type and back to its original type yields the original pointer value.

   [Note 7: Converting a pointer of type “pointer to T1” that points to an object of type T1 to the type “pointer to T2” (where T2 is an object type and the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value. — end note]

Converting a function pointer to an object pointer type or vice versa is conditionally-supported. The meaning of such a conversion is implementation-defined, except that if an implementation supports conversions in both directions, converting a prvalue of one type to the other type and back, possibly with different cv-qualification, shall yield the original pointer value.

7 The null pointer value (6.8.3) is converted to the null pointer value of the destination type.

   [Note 8: A null pointer constant of type std::nullptr_t cannot be converted to a pointer type, and a null pointer constant of integral type is not necessarily converted to a null pointer value. — end note]

A prvalue of type “pointer to member of X of type T1” can be explicitly converted to a prvalue of a different type “pointer to member of Y of type T2” if T1 and T2 are both function types or both object types. The null pointer value (7.3.13) is converted to the null pointer value of the destination type. The result of this conversion is unspecified, except in the following cases:

(10.1) — Converting a prvalue of type “pointer to member function” to a different pointer-to-member-function type and back to its original type yields the original pointer-to-member value.

---

68) The types can have different cv-qualifiers, subject to the overall restriction that a reinterpret_cast cannot cast away constness.

69) T1 and T2 can have different cv-qualifiers, subject to the overall restriction that a reinterpret_cast cannot cast away constness.
Converting a prvalue of type “pointer to data member of \( X \) of type \( T_1 \)” to the type “pointer to data member of \( Y \) of type \( T_2 \)” (where the alignment requirements of \( T_2 \) are no stricter than those of \( T_1 \)) and back to its original type yields the original pointer-to-member value.

A glvalue of type \( T_1 \), designating an object \( x \), can be cast to the type “reference to \( T_2 \)” if an expression of type “pointer to \( T_1 \)” can be explicitly converted to the type “pointer to \( T_2 \)” using a \texttt{reinterpret_cast}. The result is that of \texttt{reinterpret_cast<
T2
*>(p)} where \( p \) is a pointer to \( x \) of type “pointer to \( T_1 \)”.

Temporary is created, no copy is made, and no constructors (11.4.5) or conversion functions (11.4.8) are called.

### 7.6.1.11 Const cast

The result of the expression \texttt{const_cast<T>(v)} is of type \( T \). If \( T \) is an lvalue reference to object type, the result is an lvalue; if \( T \) is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) standard conversions are performed on the expression \( v \). Conversions that can be performed explicitly using \texttt{const_cast} are listed below. No other conversion shall be performed explicitly using \texttt{const_cast}.

For two object types \( T_1 \) and \( T_2 \) (7.3.6), a prvalue of type \( T_1 \) may be explicitly converted to the type \( T_2 \) using a \texttt{const_cast} if, considering the cv-decompositions of both types, each \( P_i^1 \) is the same as \( P_i^2 \) for all \( i \). The result of a \texttt{const_cast} refers to the original entity.

```cpp
typedef int *A[3];           // array of 3 pointer to int
typedef const int *const CA[3]; // array of 3 const pointer to const int

CA &&r = A[0];               // OK, reference binds to temporary array object
CA &&r1 = const_cast<CA>(CA); // after qualification conversion to type CA
A &&r2 = const_cast<A&&>(CA()); // error: temporary array decayed to pointer
A &&r3 = const_cast<A&&>(CA[0]); // OK

// end example]
```

For two object types \( T_1 \) and \( T_2 \), if a pointer to \( T_1 \) can be explicitly converted to the type “pointer to \( T_2 \)” using a \texttt{const_cast}, then the following conversions can also be made:

- an lvalue of type \( T_1 \) can be explicitly converted to an lvalue of type \( T_2 \) using the cast \texttt{const_cast<T2&>};
- a glvalue of type \( T_1 \) can be explicitly converted to an xvalue of type \( T_2 \) using the cast \texttt{const_cast<T2&&>};
- if \( T_1 \) is a class type, a prvalue of type \( T_1 \) can be explicitly converted to an xvalue of type \( T_2 \) using the cast \texttt{const_cast<T2&&>}.

The result of a reference \texttt{const_cast} refers to the original object if the operand is a glvalue and to the result of applying the temporary materialization conversion (7.3.5) otherwise.

A null pointer value (6.8.3) is converted to the null pointer value of the destination type. The null member pointer value (7.3.13) is converted to the null member pointer value of the destination type.

A conversion from a type \( T_1 \) to a type \( T_2 \) casts away constness if \( T_1 \) and \( T_2 \) are different, there is a cv-decomposition (7.3.6) of \( T_1 \) yielding \( n \) such that \( T_2 \) has a cv-decomposition of the form

\[
   cv_0^2 \ P_0^2 \ cv_1^1 \ P_1^1 \ \cdots \ cv_{n-1}^1 \ P_{n-1}^2 \ cv_n^2 \ U_2,
\]

and there is no qualification conversion that converts \( T_1 \) to

\[
   cv_0^2 \ P_0^1 \ cv_1^1 \ P_1^1 \ \cdots \ cv_{n-1}^1 \ P_{n-1}^1 \ cv_n^1 \ U_1.
\]

Casting from an lvalue of type \( T_1 \) to an lvalue of type \( T_2 \) using an lvalue reference cast or casting from an expression of type \( T_1 \) to an xvalue of type \( T_2 \) using an rvalue reference cast casts away constness if a cast from a prvalue of type “pointer to \( T_1 \)” to the type “pointer to \( T_2 \)” casts away constness.

---

70) This is sometimes referred to as a type pun when the result refers to the same object as the source glvalue.

71) \texttt{const_cast} is not limited to conversions that cast away a const-qualifier.
9 [Note 3: Some conversions which involve only changes in cv-qualification cannot be done using const_cast. For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior. For the same reasons, conversions between pointers to member functions, and in particular, the conversion from a pointer to a const member function to a pointer to a non-const member function, are not covered. — end note]

7.6.2 Unary expressions [expr.unary]

7.6.2.1 General [expr.unary.general]

Expressions with unary operators group right-to-left.

\[\text{unary-expression: posterior-expression}
\text{unary-operator cast-expression}
++ cast-expression
-- cast-expression
await-expression
sizeof \text{ unary-expression}
sizeof ( \text{ type-id })
sizeof ... ( \text{ identifier })
alignof ( \text{ type-id })
noexcept-expression
new-expression
delete-expression
\]

\[\text{unary-operator: one of}
* & + - ! ~\]

7.6.2.2 Unary operators [expr.unary.op]

1 The unary \text{\ast} operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is “pointer to T”, the type of the result is “T”.

[Note 1: Indirection through a pointer to an incomplete type (other than cv void) is valid. The lvalue thus obtained can be used in limited ways (to initialize a reference, for example); this lvalue must not be converted to a prvalue, see 7.3.2. — end note]

2 The result of each of the following unary operators is a prvalue.

3 The result of the unary \text{&} operator is a pointer to its operand.

\[\text{(3.1)}\]
— If the operand is a qualified-id naming a non-static or variant member \text{m} of some class \text{C} with type \text{T}, the result has type “pointer to member of class \text{C} of type \text{T}” and is a prvalue designating \text{C::m}.

\[\text{(3.2)}\]
— Otherwise, if the operand is an lvalue of type \text{T}, the resulting expression is a prvalue of type “pointer to \text{T}” whose result is a pointer to the designated object (6.7.1) or function.

[Note 2: In particular, taking the address of a variable of type “cv T” yields a pointer of type “pointer to cv T”. — end note]

\[\text{(3.3)}\]
— Otherwise, the program is ill-formed.

[Example 1:

\text{struct A \{ int i; \} ;
struct B : A \{ \} ;
... \
&B::i ... \quad \text{// has type int A::*}
\int a;
\int* \ p1 = \ &a;
\int* \ p2 = \ p1 + 1; \quad \text{// defined behavior}
\bool \ b = \ p2 > \ p1; \quad \text{// defined behavior, with value true}
— end example]

[Note 3: A pointer to member formed from a mutable non-static data member (9.2.2) does not reflect the mutable specifier associated with the non-static data member. — end note]

4 A pointer to member is only formed when an explicit \& is used and its operand is a qualified-id not enclosed in parentheses.

[Note 4: That is, the expression \&(qualified-id), where the qualified-id is enclosed in parentheses, does not form an expression of type “pointer to member”. Neither does qualified-id, because there is no implicit conversion from a

§ 7.6.2.2 127
qualified-id for a non-static member function to the type “pointer to member function” as there is from an lvalue of function type to the type “pointer to function” (7.3.4). Nor is `unqualified-id` a pointer to member, even within the scope of the unqualified-id’s class. — end note]

5 If `&` is applied to an lvalue of incomplete class type and the complete type declares `operator&()` , it is unspecified whether the operator has the built-in meaning or the operator function is called. The operand of `&` shall not be a bit-field.

[Note 5: The address of an overloaded function (Clause 12) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 12.5). Since the context might determine whether the operand is a static or non-static member function, the context can also affect whether the expression has type “pointer to function” or “pointer to member function”. — end note]

7 The operand of the unary `+` operator shall have arithmetic, unscoped enumeration, or pointer type and the result is the value of the argument. Integral promotion is performed on integral or enumeration operands. The type of the result is the type of the promoted operand.

8 The operand of the unary `-` operator shall have arithmetic or unscoped enumeration type and the result is the negative of its operand. Integral promotion is performed on integral or enumeration operands. The negative of an unsigned quantity is computed by subtracting its value from $2^n$, where $n$ is the number of bits in the promoted operand. The type of the result is the type of the promoted operand.

9 The operand of the logical negation operator `!` is contextually converted to `bool` (7.3); its value is `true` if the converted operand is `false` and `false` otherwise. The type of the result is `bool`.

10 The operand of `-` shall have integral or unscoped enumeration type; the result is the ones’ complement of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand. There is an ambiguity in the grammar when `-` is followed by a `type-name` or `decltype-specifier`. The ambiguity is resolved by treating `-` as the unary complement operator rather than as the start of an `unqualified-id` naming a destructor.

[Note 6: Because the grammar does not permit an operator to follow the `,`, `->`, or `::` tokens, a `-` followed by a `type-name` or `decltype-specifier` in a member access expression or `qualified-id` is unambiguously parsed as a destructor name. — end note]

### 7.6.2.3 Increment and decrement

[expr.pre.incr]

1 The operand of prefix `++` is modified (3.1) by adding 1. The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type other than `cv bool`, or a pointer to a completely-defined object type. An operand with volatile-qualified type is deprecated; see D.6. The result is the updated operand; it is an lvalue, and it is a bit-field if the operand is a bit-field. The expression `++x` is equivalent to `x+=1`.

[Note 1: See the discussions of addition (7.6.6) and assignment operators (7.6.19) for information on conversions. — end note]

2 The operand of prefix `--` is modified (3.1) by subtracting 1. The requirements on the operand of prefix `--` and the properties of its result are otherwise the same as those of prefix `++`.

[Note 2: For postfix increment and decrement, see 7.6.16. — end note]

### 7.6.2.4 Await

[expr.await]

1 The `co_await` expression is used to suspend evaluation of a coroutine (9.5.4) while awaiting completion of the computation represented by the operand expression.

```
await-expression:
  co_await cast-expression
```

2 An `await-expression` shall appear only in a potentially-evaluated expression within the `compound-statement` of a `function-body` outside of a `handler` (14.1). In a `declaration-statement` or in the `simple-declaration` (if any) of an `init-statement`, an `await-expression` shall appear only in an `initializer` of that `declaration-statement` or `simple-declaration`. An `await-expression` shall not appear in a default argument (9.3.4.7). An `await-expression` shall not appear in the initializer of a block-scoped variable with static or thread storage duration. A context within a function where an `await-expression` can appear is called a `suspension context` of the function.

3 Evaluation of an `await-expression` involves the following auxiliary types, expressions, and objects:

```
(3.1) — p is an lvalue naming the promise object (9.5.4) of the enclosing coroutine and P is the type of that object.
```
The `await-expression` has the same type and value category as the `await-resume` expression.

The `await-expression` evaluates the (possibly-converted) `o` expression and the `await-ready` expression, then:

1. If the result of `await-ready` is `false`, the coroutine is considered suspended. Then:
   1.1. If the type of `await-suspend` is `std::coroutine_handle<Z>`, `await-suspend.resume()` is evaluated.  
   [Note 1: This resumes the coroutine referred to by the result of `await-suspend`. Any number of coroutines can be successively resumed in this fashion, eventually returning control flow to the current coroutine caller or resumer (9.5.4). — end note]
   1.2. Otherwise, if the type of `await-suspend` is `bool`, `await-suspend` is evaluated, and the coroutine is resumed if the result is `false`.
   1.3. Otherwise, `await-suspend` is evaluated.  

   If the evaluation of `await-suspend` exits via an exception, the exception is caught, the coroutine is resumed, and the exception is immediately re-thrown (14.2). Otherwise, control flow returns to the current coroutine caller or resumer (9.5.4) without exiting any scopes (8.7).

2. If the result of `await-ready` is `true`, or when the coroutine is resumed, the `await-resume` expression is evaluated, and its result is the result of the `await-expression`.

Example 1:

```cpp
template <typename T>
struct my_future {
    /* ... */
    bool await_ready();
    void await_suspend(std::coroutine_handle<>);
    T await_resume();
};

template <class Rep, class Period>
auto operator co_await(std::chrono::duration<Rep, Period> d) {
    struct awaiter {
        std::chrono::system_clock::duration duration;
        /* ... */
        awaiter(std::chrono::system_clock::duration d) : duration(d) {}  
        bool await_ready() const { return duration.count() <= 0; }
        void await_resume() {}
        void await_suspend(std::coroutine_handle<> h) { /* ... */ }
    };
    return awaiter(d);
}

using namespace std::chrono;

my_future<int> h();
```
my_future<void> g() {
    std::cout << "just about go to sleep...\n";
    co_await 10ms;
    std::cout << "resumed\n";
    co_await h();
}

auto f(int x = co_await h());       // error: await-expression outside of function suspension context
int a[] = { co_await h() };         // error: await-expression outside of function suspension context

— end example]  

7.6.2.5 Sizeof [expr.sizeof]

The sizeof operator yields the number of bytes occupied by a non-potentially-overlapping object of the type of its operand. The operand is either an expression, which is an unevaluated operand (7.2), or a parenthesized type-id. The sizeof operator shall not be applied to an expression that has function or incomplete type, to the parenthesized name of such types, or to a glvalue that designates a bit-field. The result of sizeof applied to any of the narrow character types is 1. The result of sizeof applied to any other fundamental type (6.8.2) is implementation-defined.

[Note 1: In particular, the values of sizeof(bool), sizeof(char16_t), sizeof(char32_t), and sizeof(wchar_t) are implementation-defined. — end note]

[Note 2: See 6.7.1 for the definition of byte and 6.8 for the definition of object representation. — end note]

2 When applied to a reference type, the result is the size of the referenced type. When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array. The result of applying sizeof to a potentially-overlapping subobject is the size of the type, not the size of the subobject. When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of n elements is n times the size of an element.

3 The lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) standard conversions are not applied to the operand of sizeof. If the operand is a prvalue, the temporary materialization conversion (7.3.5) is applied.

4 The identifier in a sizeof... expression shall name a pack. The sizeof... operator yields the number of elements in the pack (13.7.4). A sizeof... expression is a pack expansion (13.7.4).

[Example 1:

    template<class... Types>
    struct count {
        static const std::size_t value = sizeof...(Types);
    };

— end example]

5 The result of sizeof and sizeof... is a prvalue of type std::size_t.

[Note 3: A sizeof expression is an integral constant expression (7.7). The type std::size_t is defined in the standard header <cstdlib> (17.2.1, 17.2.4). — end note]

7.6.2.6 Alignof [expr.alignof]

An alignof expression yields the alignment requirement of its operand type. The operand shall be a type-id representing a complete object type, or an array thereof, or a reference to one of those types.

2 The result is a prvalue of type std::size_t.

[Note 1: An alignof expression is an integral constant expression (7.7). The type std::size_t is defined in the standard header <cstdlib> (17.2.1, 17.2.4). — end note]

3 When alignof is applied to a reference type, the result is the alignment of the referenced type. When alignof is applied to an array type, the result is the alignment of the element type.

72) sizeof(bool) is not required to be 1.
73) The actual size of a potentially-overlapping subobject can be less than the result of applying sizeof to the subobject, due to virtual base classes and less strict padding requirements on potentially-overlapping subobjects.
7.6.2.7 noexcept operator

The noexcept operator determines whether the evaluation of its operand, which is an unevaluated operand (7.2), can throw an exception (14.2).

noexcept-expression:
  noexcept ( expression )

The result of the noexcept operator is a prvalue of type bool.

[Note 1: A noexcept-expression is an integral constant expression (7.7). — end note]

3 The result of the noexcept operator is true unless the expression is potentially-throwing (14.5).

7.6.2.8 New

The new-expression attempts to create an object of the type-id (9.3.2) or new-type-id to which it is applied. The type of that object is the allocated type. This type shall be a complete object type, but not an abstract class type or array thereof (6.7.2, 6.8, 11.7.4).

[Note 1: Because references are not objects, references cannot be created by new-expressions. — end note]

[Note 2: The type-id can be a cv-qualified type, in which case the object created by the new-expression has a cv-qualified type. — end note]

e new-expression:
  ::opt new new-placementopt new-type-id opt new-initializeropt

new-placement: ( expression-list )

new-type-id: type-specifier-seq new-declaratoropt

new-declarator:
  ptr-operator new-declaratoropt
  noptr-new-declarator

noptr-new-declarator: [ expressionopt ] attribute-specifier-seqopt
  [ constant-expression ] attribute-specifier-seqopt

new-initializer:
  ( expression-listopt )
  braced-init-list

If a placeholder type (9.2.9.6) appears in the type-specifier-seq of a new-type-id or type-id of a new-expression, the allocated type is deduced as follows: Let init be the new-initializer, if any, and T be the new-type-id or type-id of the new-expression, then the allocated type is the type deduced for the variable x in the invented declaration (9.2.9.6):

T x init;

[Example 1:
  new auto(1);        // allocated type is int
  auto x = new auto('a');  // allocated type is char, x is of type char*
]

template<class T> struct A { A(T, T); }; auto y = new A(1, 2);  // allocated type is A<int>

[Note 3: This prevents ambiguities between the declarator operators k, kk, *, and [] and their expression counterparts. — end note]

[Example 2:
  new int * i;        // syntax error: parsed as (new int*) i, not as (new int)*i

The * is the pointer declarator and not the multiplication operator. — end example]

[Note 4: Parentheses in a new-type-id of a new-expression can have surprising effects.

[Example 3:
  new int(*[10])();   // error

§ 7.6.2.8 131
is ill-formed because the binding is

\[
\text{new int} (*[10])();
\]  // error

Instead, the explicitly parenthesized version of the `new` operator can be used to create objects of compound types (6.8.3):

\[
\text{new} (\text{int} (*[10])());
\]

allocates an array of 10 pointers to functions (taking no argument and returning `int`). — end example]

—end note]

5 Objects created by a `new-expression` have dynamic storage duration (6.7.5.5).

[Note 5: The lifetime of such an object is not necessarily restricted to the scope in which it is created. — end note]

When the allocated object is not an array, the result of the `new-expression` is a pointer to the object created.

6 When the allocated object is an array (that is, the `noptr-new-declarator` syntax is used or the `new-type-id` or `type-id` denotes an array type), the `new-expression` yields a pointer to the initial element (if any) of the array.

[Note 6: Both `new int` and `new int[10]` have type `int*` and the type of `new int[1][10]` is `int (*)&[10]` — end note]

The `attribute-specifier-seq` in a `noptr-new-declarator` appertains to the associated array type.

7 Every `constant-expression` in a `noptr-new-declarator` shall be a converted constant expression (7.7) of type `std::size_t` and its value shall be greater than zero.

[Example 4: Given the definition `int n = 42, new float[n][5]` is well-formed (because `n` is the `expression` of a `noptr-new-declarator`), but `new float[5][n]` is ill-formed (because `n` is not a constant expression). — end example]

8 If the `type-id` or `new-type-id` denotes an array type of unknown bound (9.3.4.5), the `new-initializer` shall not be omitted; the allocated object is an array with `n` elements, where `n` is determined from the number of initial elements supplied in the `new-initializer` (9.4.2, 9.4.3).

9 If the `expression` in a `noptr-new-declarator` is present, it is implicitly converted to `std::size_t`. The `expression` is erroneous if:

\[(9.1)\] — the expression is of non-class type and its value before converting to `std::size_t` is less than zero;

\[(9.2)\] — the expression is of class type and its value before application of the second standard conversion (12.4.4.2.3)\(^{74}\) is less than zero;

\[(9.3)\] — its value is such that the size of the allocated object would exceed the implementation-defined limit (Annex B); or

\[(9.4)\] — the `new-initializer` is a `braced-init-list` and the number of array elements for which initializers are provided (including the terminating `\'0\'` in a `string-literal` (5.13.5)) exceeds the number of elements to initialize.

If the `expression` is erroneous after converting to `std::size_t`:

\[(9.5)\] — if the `expression` is a core constant expression, the program is ill-formed;

\[(9.6)\] — otherwise, an allocation function is not called; instead

\[(9.6.1)\] — if the allocation function that would have been called has a non-throwing exception specification (14.5), the value of the `new-expression` is the null pointer value of the required result type;

\[(9.6.2)\] — otherwise, the `new-expression` terminates by throwing an exception of a type that would match a handler (14.4) of type `std::bad_array_new_length` (17.6.4.2).

When the value of the `expression` is zero, the allocation function is called to allocate an array with no elements.

10 A `new-expression` may obtain storage for the object by calling an allocation function (6.7.5.5.2). If the `new-expression` terminates by throwing an exception, it may release storage by calling a deallocation function (6.7.5.5.3). If the allocated type is a non-array type, the allocation function’s name is `operator new` and the deallocation function’s name is `operator delete`. If the allocated type is an array type, the allocation function’s name is `operator new[]` and the deallocation function’s name is `operator delete[]`.

[Note 7: An implementation is required to provide default definitions for the global allocation functions (6.7.5.5, 17.6.3.2, 17.6.3.3). A C++ program can provide alternative definitions of these functions (16.4.5.6) and/or class-specific versions (11.12). The set of allocation and deallocation functions that can be called by a `new-expression` could include functions that do not perform allocation or deallocation; for example, see 17.6.3.4. — end note]

\(^{74}\) If the conversion function returns a signed integer type, the second standard conversion converts to the unsigned type `std::size_t` and thus thwarts any attempt to detect a negative value afterwards.
If the `new-expression` begins with a unary `::` operator, the allocation function’s name is looked up in the global scope. Otherwise, if the allocated type is a class type `T` or array thereof, the allocation function’s name is looked up in the scope of `T`. If this lookup fails to find the name, or if the allocated type is not a class type, the allocation function’s name is looked up in the global scope.

An implementation is allowed to omit a call to a replaceable global allocation function (17.6.3.2, 17.6.3.3). When it does so, the storage is instead provided by the implementation or provided by extending the allocation of another `new-expression`.

During an evaluation of a constant expression, a call to an allocation function is always omitted.

[Note 8: Only `new-expressions` that would otherwise result in a call to a replaceable global allocation function can be evaluated in constant expressions (7.7). — end note]

The implementation may extend the allocation of a `new-expression e1` to provide storage for a `new-expression e2` if the following would be true were the allocation not extended:

1. The evaluation of `e1` is sequenced before the evaluation of `e2`, and
2. `e2` is evaluated whenever `e1` obtains storage, and
3. Both `e1` and `e2` invoke the same replaceable global allocation function, and
4. If the allocation function invoked by `e1` and `e2` is throwing, any exceptions thrown in the evaluation of either `e1` or `e2` would be first caught in the same handler, and
5. The pointer values produced by `e1` and `e2` are operands to evaluated `delete-expressions`, and
6. The evaluation of `e2` is sequenced before the evaluation of the `delete-expression` whose operand is the pointer value produced by `e1`.

[Example 5:]
```cpp
void can_merge(int x) {
    // These allocations are safe for merging:
    std::unique_ptr<char[]> a(new (std::nothrow) char[8]);
    std::unique_ptr<char[]> b(new (std::nothrow) char[8]);
    std::unique_ptr<char[]> c(new (std::nothrow) char[x]);
    g(a.get(), b.get(), c.get());
}

void cannot_merge(int x) {
    std::unique_ptr<char[]> a(new char[8]);
    try {
        // Merging this allocation would change its catch handler.
        std::unique_ptr<char[]> b(new char[x]);
    } catch (const std::bad_alloc& e) {
        std::cerr << "Allocation failed: " << e.what() << std::endl;
        throw;
    }
}
```

When a `new-expression` calls an allocation function and that allocation has not been extended, the `new-expression` passes the amount of space requested to the allocation function as the first argument of type `std::size_t`. That argument shall be no less than the size of the object being created; it may be greater than the size of the object being created only if the object is an array and the allocation function is not a non-allocating form (17.6.3.4). For arrays of `char`, `unsigned char`, and `std::byte`, the difference between the result of the `new-expression` and the address returned by the allocation function shall be an integral multiple of the strictest fundamental alignment requirement (6.7.6) of any object type whose size is no greater than the size of the array being created.

[Note 9: Because allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type with fundamental alignment, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. — end note]

When a `new-expression` calls an allocation function and that allocation has been extended, the size argument to the allocation call shall be no greater than the sum of the sizes for the omitted calls as specified above,
plus the size for the extended call had it not been extended, plus any padding necessary to align the allocated objects within the allocated memory.

The new-placement syntax is used to supply additional arguments to an allocation function; such an expression is called a placement new-expression.

Overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and has type std::size_t. If the type of the allocated object has new-extended alignment, the next argument is the type’s alignment, and has type std::align_val_t. If the new-placement syntax is used, the initializer-clauses in its expression-list are the succeeding arguments. If no matching function is found then

(18.1) if the allocated object type has new-extended alignment, the alignment argument is removed from the argument list;

(18.2) otherwise, an argument that is the type’s alignment and has type std::align_val_t is added into the argument list immediately after the first argument;

and then overload resolution is performed again.

[Example 6:

(19.1) new T results in one of the following calls:
  operator new(sizeof(T))
  operator new(sizeof(T), std::align_val_t(alignof(T)))

(19.2) new(2,f) T results in one of the following calls:
  operator new(sizeof(T), 2, f)
  operator new(sizeof(T), std::align_val_t(alignof(T)), 2, f)

(19.3) new T[5] results in one of the following calls:
  operator new[] (sizeof(T) + 5 + x)
  operator new[] (sizeof(T) + 5 + x, std::align_val_t(alignof(T)))

(19.4) new(2,f) T[5] results in one of the following calls:
  operator new[] (sizeof(T) + 5 + x, 2, f)
  operator new[] (sizeof(T) + 5 + x, std::align_val_t(alignof(T)), 2, f)

Here, each instance of x is a non-negative unspecified value representing array allocation overhead; the result of the new-expression will be offset by this amount from the value returned by operator new[]. This overhead may be applied in all array new-expressions, including those referencing a placement allocation function, except when referencing the library function operator new[](std::size_t, void*). The amount of overhead may vary from one invocation of new to another. —end example]

Note 10: Unless an allocation function has a non-throwing exception specification (14.5), it indicates failure to allocate storage by throwing a std::bad_alloc exception (6.7.5.5.2, Clause 14, 17.6.4.1); it returns a non-null pointer otherwise. If the allocation function has a non-throwing exception specification, it returns null to indicate failure to allocate storage and a non-null pointer otherwise. —end note]

If the allocation function is a non-allocating form (17.6.3.4) that returns null, the behavior is undefined. Otherwise, if the allocation function returns null, initialization shall not be done, the deallocation function shall not be called, and the value of the new-expression shall be null.

Note 11: When the allocation function returns a value other than null, it must be a pointer to a block of storage in which space for the object has been reserved. The block of storage is assumed to be appropriately aligned and of the requested size. The address of the created object will not necessarily be the same as that of the block if the object is an array. —end note]

A new-expression that creates an object of type T initializes that object as follows:

(22.1) If the new-initializer is omitted, the object is default-initialized (9.4).

[Note 12: If no initialization is performed, the object has an indeterminate value. —end note]

(22.2) Otherwise, the new-initializer is interpreted according to the initialization rules of 9.4 for direct-initialization.

The invocation of the allocation function is sequenced before the evaluations of expressions in the new-initializer. Initialization of the allocated object is sequenced before the value computation of the new-expression.

If the new-expression creates an object or an array of objects of class type, access and ambiguity control are done for the allocation function, the deallocation function (11.12), and the constructor (11.4.5) selected for § 7.6.2.8
the initialization (if any). If the new-expression creates an array of objects of class type, the destructor is potentially invoked (11.4.7).

If any part of the object initialization described above terminates by throwing an exception and a suitable deallocation function can be found, the deallocation function is called to free the memory in which the object was being constructed, after which the exception continues to propagate in the context of the new-expression. If no unambiguous matching deallocation function can be found, propagating the exception does not cause the object’s memory to be freed.

[Note 13: This is appropriate when the called allocation function does not allocate memory; otherwise, it is likely to result in a memory leak. — end note]

If the new-expression begins with a unary :: operator, the deallocation function’s name is looked up in the global scope. Otherwise, if the allocated type is a class type T or an array thereof, the deallocation function’s name is looked up in the scope of T. If this lookup fails to find the name, or if the allocated type is not a class type or array thereof, the deallocation function’s name is looked up in the global scope.

A declaration of a placement deallocation function matches the declaration of a placement allocation function if it has the same number of parameters and, after parameter transformations (9.3.4.6), all parameter types except the first are identical. If the lookup finds a single matching deallocation function, that function will be called; otherwise, no deallocation function will be called. If the lookup finds a usual deallocation function and that function, considered as a placement deallocation function, would have been selected as a match for the allocation function, the program is ill-formed. For a non-placement allocation function, the normal deallocation function lookup is used to find the matching deallocation function (7.6.2.9).

[Example 7:]

```c++
struct S {
    // Placement allocation function:
    static void* operator new(std::size_t, std::size_t);

    // Usual (non-placement) deallocation function:
    static void operator delete(void*, std::size_t);
};
S* p = new (0) S;  // error: non-placement deallocation function matches
                  // placement allocation function
                  // - end example]
```

If a new-expression calls a deallocation function, it passes the value returned from the allocation function call as the first argument of type void*. If a placement deallocation function is called, it is passed the same additional arguments as were passed to the placement allocation function, that is, the same arguments as those specified with the new-placement syntax. If the implementation is allowed to introduce a temporary object or make a copy of any argument as part of the call to the allocation function, it is unspecified whether the same object is used in the call to both the allocation and deallocation functions.

7.6.2.9 Delete [expr.delete]

The delete-expression operator destroys a most derived object (6.7.2) or array created by a new-expression.

```c++
delete-expression:
    ::opt delete cast-expression
    ::opt delete [] cast-expression
```

The first alternative is a single-object delete expression, and the second is an array delete expression. Whenever the delete keyword is immediately followed by empty square brackets, it shall be interpreted as the second alternative. If a lambda-expression with a lambda-introducer that consists of empty square brackets can follow the delete keyword if the lambda-expression is enclosed in parentheses. This implies that an object cannot be deleted using a pointer of type void* because void is not an object type.
value, a pointer to a non-array object created by a previous `new-expression`, or a pointer to a subobject (6.7.2) representing a base class of such an object (11.7). If not, the behavior is undefined. In an array delete expression, the value of the operand of `delete` may be a null pointer value or a pointer value that resulted from a previous array `new-expression`.\(^{78}\) If not, the behavior is undefined.

[Note 1: This means that the syntax of the `delete-expression` must match the type of the object allocated by `new`, not the syntax of the `new-expression`. — end note]

[Note 2: A pointer to a `const` type can be the operand of a `delete-expression`; it is not necessary to cast away the constness (7.6.1.11) of the pointer expression before it is used as the operand of the `delete-expression`. — end note]

3 In a single-object delete expression, if the static type of the object to be deleted is different from its dynamic type and the selected deallocation function (see below) is not a destroying operator delete, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined. In an array delete expression, if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.

4 The `cast-expression` in a `delete-expression` shall be evaluated exactly once.

5 If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined.

6 If the value of the operand of the `delete-expression` is not a null pointer value and the selected deallocation function (see below) is not a destroying operator delete, the `delete-expression` will invoke the destructor (if any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, in reverse order of the completion of their constructor; see 11.10.3).

7 If the value of the operand of the `delete-expression` is not a null pointer value, then:

\[
\text{(1.1)} \quad \text{If the allocation call for the `new-expression` for the object to be deleted was not omitted and the allocation was not extended (7.6.2.8), the `delete-expression` shall call a deallocation function (6.7.5.5.3). The value returned from the allocation call of the `new-expression` shall be passed as the first argument to the deallocation function.}
\]

\[
\text{(1.2)} \quad \text{Otherwise, if the allocation was extended or was provided by extending the allocation of another `new-expression`, and the `delete-expression` for every other pointer value produced by a `new-expression` that had storage provided by the extended `new-expression` has been evaluated, the `delete-expression` shall call a deallocation function. The value returned from the allocation call of the extended `new-expression` shall be passed as the first argument to the deallocation function.}
\]

\[
\text{(1.3)} \quad \text{Otherwise, the `delete-expression` will not call a deallocation function.}
\]

[Note 3: The deallocation function is called regardless of whether the destructor for the object or some element of the array throws an exception. — end note]

If the value of the operand of the `delete-expression` is a null pointer value, it is unspecified whether a deallocation function will be called as described above.

8 [Note 4: An implementation provides default definitions of the global deallocation functions `operator delete` for non-arrays (17.6.3.2) and `operator delete[]` for arrays (17.6.3.3). A C++ program can provide alternative definitions of these functions (16.4.5.6), and/or class-specific versions (11.12). — end note]

9 When the keyword `delete` in a `delete-expression` is preceded by the unary `::` operator, the deallocation function’s name is looked up in global scope. Otherwise, the lookup considers class-specific deallocation functions (11.12). If no class-specific deallocation function is found, the deallocation function’s name is looked up in global scope.

10 If deallocation function lookup finds more than one usual deallocation function, the function to be called is selected as follows:

\[
\text{(10.1)} \quad \text{If any of the deallocation functions is a destroying operator delete, all deallocation functions that are not destroying operator deletes are eliminated from further consideration.}
\]

\[
\text{(10.2)} \quad \text{If the type has new-extended alignment, a function with a parameter of type `std::align_val_t` is preferred; otherwise a function without such a parameter is preferred. If any preferred functions are found, all non-preferred functions are eliminated from further consideration.}
\]

\(^{78}\) For nonzero-length arrays, this is the same as a pointer to the first element of the array created by that `new-expression`. Zero-length arrays do not have a first element.
If exactly one function remains, that function is selected and the selection process terminates.

If the deallocation functions have class scope, the one without a parameter of type `std::size_t` is selected.

If the type is complete and if, for an array delete expression only, the operand is a pointer to a class type with a non-trivial destructor or a (possibly multi-dimensional) array thereof, the function with a parameter of type `std::size_t` is selected.

Otherwise, it is unspecified whether a deallocation function with a parameter of type `std::size_t` is selected.

For a single-object delete expression, the deleted object is the object denoted by the operand if its static type does not have a virtual destructor, and its most-derived object otherwise.

For an array delete expression, the deleted object is the array object. When a `delete-expression` is executed, the selected deallocation function shall be called with the address of the deleted object in a single-object delete expression, or the address of the deleted object suitably adjusted for the array allocation overhead (7.6.2.8) in an array delete expression, as its first argument.

If a destroying operator delete is used, an unspecified value is passed as the argument corresponding to the parameter of type `std::destroying_delete_t`. If a deallocation function with a parameter of type `std::align_val_t` is used, the alignment of the type of the deleted object is passed as the corresponding argument. If a deallocation function with a parameter of type `std::size_t` is used, the size of the deleted object in a single-object delete expression, or of the array plus allocation overhead in an array delete expression, is passed as the corresponding argument.

Access and ambiguity control are done for both the deallocation function and the destructor (11.4.7, 11.12).

**7.6.3 Explicit type conversion (cast notation)**

The result of the expression `(T) cast-expression` is of type T. The result is an lvalue if T is an lvalue reference type or an rvalue reference to function type and an xvalue if T is an rvalue reference to object type; otherwise the result is a prvalue.

Any type conversion not mentioned below and not explicitly defined by the user (11.4.8) is ill-formed.

The conversions performed by

- a `const_cast` (7.6.1.11),
- a `static_cast` (7.6.1.9),
- a `static_cast` followed by a `const_cast`,
- a `reinterpret_cast` (7.6.1.10), or
- a `reinterpret_cast` followed by a `const_cast`,

can be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply, with the exception that in performing a `static_cast` in the following situations the conversion is valid even if the base class is inaccessible:

- a pointer to an object of derived class type or an lvalue or rvalue of derived class type may be explicitly converted to a pointer or reference to an unambiguous base class type, respectively.
— a pointer to member of derived class type may be explicitly converted to a pointer to member of an
unambiguous non-virtual base class type;

— a pointer to an object of an unambiguous non-virtual base class type, a glvalue of an unambiguous
non-virtual base class type, or a pointer to member of an unambiguous non-virtual base class type
may be explicitly converted to a pointer, a reference, or a pointer to member of a derived class type,
respectively.

If a conversion can be interpreted in more than one of the ways listed above, the interpretation that appears
first in the list is used, even if a cast resulting from that interpretation is ill-formed. If a conversion can be
interpreted in more than one way as a static_cast followed by a const_cast, the conversion is ill-formed.

[Example 1:

```cpp
struct A { }
struct I1 : A { }
struct I2 : A { }
struct D : I1, I2 { }
A* foo( D* p ) {
    return (A*)( p );
    // ill-formed static_cast interpretation
}
```
— end example]

Let an operand of a cast using the cast notation can be a prvalue of type “pointer to incomplete class type”.
The destination type of a cast using the cast notation can be “pointer to incomplete class type”. If both the
operand and destination types are class types and one or both are incomplete, it is unspecified whether the
static_cast or the reinterpret_cast interpretation is used, even if there is an inheritance relationship
between the two classes.

[Note 1: For example, if the classes were defined later in the translation unit, a multi-pass compiler would be permitted
to interpret a cast between pointers to the classes as if the class types were complete at the point of the cast. — end
note]

### 7.6.4 Pointer-to-member operators

The pointer-to-member operators 

```cpp
pm-expression
    cast-expression
pm-expression
    pm-expression
    .* cast-expression
pm-expression
    pm-expression
    ->* cast-expression
```

2 The binary operator .* binds its second operand, which shall be of type “pointer to member of T” to its first
operand, which shall be a glvalue of class T or of a class of which T is an unambiguous and accessible base
class. The result is an object or a function of the type specified by the second operand.

3 The binary operator ->* binds its second operand, which shall be of type “pointer to member of T” to its first
operand, which shall be of type “pointer to U” where U is either T or a class of which T is an unambiguous
and accessible base class. The expression E1->*E2 is converted into the equivalent form (*((E1)).*E2.

4 Abbreviating pm-expression.*cast-expression as E1.*E2, E1 is called the object expression. If the dynamic type
of E1 does not contain the member to which E2 refers, the behavior is undefined. Otherwise, the expression
E1 is sequenced before the expression E2.

5 The restrictions on cv-qualification, and the manner in which the cv-qualifiers of the operands are combined
to produce the cv-qualifiers of the result, are the same as the rules for E1.E2 given in 7.6.1.5.

[Note 1: It is not possible to use a pointer to member that refers to a mutable member to modify a const class object.
For example,

```cpp
struct S {
    S() : i(0) { }
    mutable int i;
};
void f()
{
    const S cs;
    int S::* pm = &S::i;    // pm refers to mutable member S::i
    cs.*pm = 88;           // error: cs is a const object
}
```

§ 7.6.4
If the result of \(*\) or \(-\ast\) is a function, then that result can be used only as the operand for the function call operator ()

\[\text{Example 1:}\]
\[(\text{ptr}_\text{to}_\text{obj}->\text{ptr}_\text{to}_\text{mfct})(10);\]
calls the member function denoted by \text{ptr}_\text{to}_\text{mfct} for the object pointed to by \text{ptr}_\text{to}_\text{obj}.  — end example\]

In a \(*\) expression whose object expression is an rvalue, the program is ill-formed if the second operand is a pointer to member function whose ref-qualifier is &, unless its cv-qualifier-seq is const. In a \(*\) expression whose object expression is an lvalue, the program is ill-formed if the second operand is a pointer to member function whose ref-qualifier is &\&. The result of a \(*\) expression whose second operand is a pointer to a data member is an lvalue if the first operand is an lvalue and an xvalue otherwise. The result of a \(*\) expression whose second operand is a pointer to a member function is a prvalue. If the second operand is the null member pointer value (7.3.13), the behavior is undefined.

### 7.6.5 Multiplicative operators

The multiplicative operators \(*\), \(^\text{}/\) and \(\%\) group left-to-right.

\[
\begin{align*}
\text{multiplicative-expression} & : \text{pm-expression} \ast \text{pm-expression} \\
& \ast \text{pm-expression} \\
& \ast \text{pm-expression} / \text{pm-expression} \\
& \ast \text{pm-expression} \% \text{pm-expression}
\end{align*}
\]

The operands of \(*\) and \(^\text{/}\) shall have arithmetic or unscoped enumeration type; the operands of \(\%\) shall have integral or unscoped enumeration type. The usual arithmetic conversions (7.4) are performed on the operands and determine the type of the result.

The binary \(*\) operator indicates multiplication.

The binary \(^\text{/}\) operator yields the quotient, and the binary \(\%\) operator yields the remainder from the division of the first expression by the second. If the second operand of \(^\text{/}\) or \(\%\) is zero the behavior is undefined. For integral operands the \(^\text{/}\) operator yields the algebraic quotient with any fractional part discarded;\(^{79}\) if the quotient \(a/b\) is representable in the type of the result, \((a/b)\ast b + a\%b\) is equal to \(a\); otherwise, the behavior of both \(a/b\) and \(a\%b\) is undefined.

### 7.6.6 Additive operators

The additive operators \(+\) and \(-\) group left-to-right. The usual arithmetic conversions (7.4) are performed for operands of arithmetic or enumeration type.

\[
\begin{align*}
\text{additive-expression} & : \text{multiplicative-expression} \\
& \ast \text{multiplicative-expression} \\
& \ast \text{multiplicative-expression} \% \text{multiplicative-expression} \\
& \ast \text{multiplicative-expression} / \text{multiplicative-expression}
\end{align*}
\]

For addition, either both operands shall have arithmetic or unscoped enumeration type, or one operand shall be a pointer to a completely-defined object type and the other shall have integral or unscoped enumeration type.

For subtraction, one of the following shall hold:

1. Both operands have arithmetic or unscoped enumeration type; or
2. Both operands are pointers to cv-qualified or cv-unqualified versions of the same completely-defined object type; or
3. The left operand is a pointer to a completely-defined object type and the right operand has integral or unscoped enumeration type.

The result of the binary \(+\) operator is the sum of the operands. The result of the binary \(-\) operator is the difference resulting from the subtraction of the second operand from the first.

When an expression \(J\) that has integral type is added to or subtracted from an expression \(P\) of pointer type, the result has the type of \(P\).

\(^{79}\) This is often called truncation towards zero.
§ 7.6.7 Shift operators [expr.shift]

1 The shift operators << and >> group left-to-right.

\[
\begin{align*}
\text{shift-expression:} & \\
& \text{additive-expression} \\
& \text{shift-expression} << \text{additive-expression} \\
& \text{shift-expression} >> \text{additive-expression}
\end{align*}
\]

The operands shall be of integral or unscoped enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. The behavior is undefined if the right operand is negative, or greater than or equal to the width of the promoted left operand.

2 The value of \(E1 << E2\) is the unique value congruent to \(E1 \times 2^{E2}\) modulo \(2^N\), where \(N\) is the width of the type of the result.

[Note 1: \(E1\) is left-shifted \(E2\) bit positions; vacated bits are zero-filled. —end note]

3 The value of \(E1 >> E2\) is \(E1 / 2^{E2}\), rounded down.

[Note 2: \(E1\) is right-shifted \(E2\) bit positions. Right-shift on signed integral types is an arithmetic right shift, which performs sign-extension. —end note]

4 The expression \(E1\) is sequenced before the expression \(E2\).

§ 7.6.8 Three-way comparison operator [expr.spaceship]

1 The three-way comparison operator groups left-to-right.

\[
\begin{align*}
\text{compare-expression:} & \\
& \text{shift-expression} \\
& \text{compare-expression} <=> \text{shift-expression}
\end{align*}
\]

2 The expression \(p <=> q\) is a prvalue indicating whether \(p\) is less than, equal to, greater than, or incomparable with \(q\).

3 If one of the operands is of type \texttt{bool} and the other is not, the program is ill-formed.

4 If both operands have arithmetic types, or one operand has integral type and the other operand has unscoped enumeration type, the usual arithmetic conversions (7.4) are applied to the operands. Then:

(4.1) — If a narrowing conversion (9.4.5) is required, other than from an integral type to a floating-point type, the program is ill-formed.

80) As specified in 6.8.3, an object that is not an array element is considered to belong to a single-element array for this purpose and a pointer past the last element of an array of \(n\) elements is considered to be equivalent to a pointer to a hypothetical array element \(n\) for this purpose.
Otherwise, if the operands have integral type, the result is of type std::strong_ordering. The result is std::strong_ordering::equal if both operands are arithmetically equal, std::strong_ordering::less if the first operand is arithmetically less than the second operand, and std::strong_ordering::greater otherwise.

Otherwise, the operands have floating-point type, and the result is of type std::partial_ordering. The expression a <=> b yields std::partial_ordering::less if a is less than b, std::partial_ordering::greater if a is greater than b, std::partial_ordering::equivalent if a is equivalent to b, and std::partial_ordering::unordered otherwise.

If both operands have the same enumeration type E, the operator yields the result of converting the operands to the underlying type of E and applying <=> to the converted operands.

If at least one of the operands is of object pointer type and the other operand is of object pointer or array type, array-to-pointer conversions (7.3.3), pointer conversions (7.3.12), and qualification conversions (7.3.6) are performed on both operands to bring them to their composite pointer type (7.2.2). After the conversions, the operands shall have the same type.

[Note 1: If both of the operands are arrays, array-to-pointer conversions (7.3.3) are not applied. — end note]

In this case, p <=> q is of type std::strong_ordering and the result is defined by the following rules:

1. If two pointer operands p and q compare equal (7.6.10), p <=> q yields std::strong_ordering::equal;
2. otherwise, if p and q compare unequal, p <=> q yields std::strong_ordering::less if q compares greater than p and std::strong_ordering::greater if p compares greater than q (7.6.9);
3. otherwise, the result is unspecified.

Otherwise, the program is ill-formed.

The three comparison category types (17.11.2) (the types std::strong_ordering, std::weak_ordering, and std::partial_ordering) are not predefined; if the header <compare> (17.11.1) is not imported or included prior to a use of such a class type – even an implicit use in which the type is not named (e.g., via the auto specifier (9.2.9.6) in a defaulted three-way comparison (11.11.3) or use of the built-in operator) – the program is ill-formed.

7.6.9 Relational operators

The relational operators group left-to-right.

[Example 1: a<b<c means (a<b)<c and not (a<b)&&(b<c). — end example]

The lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) standard conversions are performed on the operands. The comparison is deprecated if both operands were of array type prior to these conversions (D.5).

The converted operands shall have arithmetic, enumeration, or pointer type. The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield false or true. The type of the result is bool.

The usual arithmetic conversions (7.4) are performed on operands of arithmetic or enumeration type. If both operands are pointers, pointer conversions (7.3.12) and qualification conversions (7.3.6) are performed to bring them to their composite pointer type (7.2.2). After conversions, the operands shall have the same type.

The result of comparing unequal pointers to objects is defined in terms of a partial order consistent with the following rules:

1. If two pointers point to different elements of the same array, or to subobjects thereof, the pointer to the element with the higher subscript is required to compare greater.

---

81) As specified in 6.8.3, an object that is not an array element is considered to belong to a single-element array for this purpose and a pointer past the last element of an array of n elements is considered to be equivalent to a pointer to a hypothetical array element n for this purpose.
If two pointers point to different non-static data members of the same object, or to subobjects of such members, recursively, the pointer to the later declared member is required to compare greater provided the two members have the same access control (11.9), neither member is a subobject of zero size, and their class is not a union.

Otherwise, neither pointer is required to compare greater than the other.

If two operands \( p \) and \( q \) compare equal (7.6.10), \( p \leq q \) and \( p = q \) both yield \( \text{true} \) and \( p < q \) and \( p > q \) both yield \( \text{false} \). Otherwise, if a pointer \( p \) compares greater than a pointer \( q \), \( p \geq q \), \( p > q \) all yield \( \text{true} \) and \( p \leq q \), \( p < q \), \( q \geq p \), and \( q > p \) all yield \( \text{false} \). Otherwise, the result of each of the operators is unspecified.

If both operands (after conversions) are of arithmetic or enumeration type, each of the operators shall yield \( \text{true} \) if the specified relationship is true and \( \text{false} \) if it is false.

**Equality operators**

\[
\begin{align*}
\text{equality-expression} : & \quad \text{relational-expression} \\
\text{equality-expression} &= \text{relational-expression} \\
\text{equality-expression} \neq \text{relational-expression}
\end{align*}
\]

1. The \( == \) (equal to) and the \( != \) (not equal to) operators group left-to-right. The lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) standard conversions are performed on the operands. The comparison is deprecated if both operands were of array type prior to these conversions (D.5).

2. The converted operands shall have arithmetic, enumeration, pointer, or pointer-to-member type, or type `std::nullptr_t`. The operators \( == \) and \( != \) both yield \( \text{true} \) or \( \text{false} \), i.e., a result of type \( \text{bool} \). In each case below, the operands shall have the same type after the specified conversions have been applied.

3. If at least one of the operands is a pointer, pointer conversions (7.3.12), function pointer conversions (7.3.14), and qualification conversions (7.3.6) are performed on both operands to bring them to their composite pointer type (7.2.2). Comparing pointers is defined as follows:

   1. If one pointer represents the address of a complete object, and another pointer represents the address one past the last element of a different complete object, the result of the comparison is unspecified.

   2. Otherwise, if the pointers are both null, both point to the same function, or both represent the same address (6.8.3), they compare equal.

   3. Otherwise, the pointers compare unequal.

4. If at least one of the operands is a pointer to member, pointer-to-member conversions (7.3.13), function pointer conversions (7.3.14), and qualification conversions (7.3.6) are performed on both operands to bring them to their composite pointer type (7.2.2). Comparing pointers to members is defined as follows:

   1. If two pointers to members are both the null member pointer value, they compare equal.

   2. If only one of two pointers to members is the null member pointer value, they compare unequal.

   3. If either is a pointer to a virtual member function, the result is unspecified.

   4. If one refers to a member of class \( C_1 \) and the other refers to a member of a different class \( C_2 \), where neither is a base class of the other, the result is unspecified.

   **Example 1:**

   ```
   struct A {}
   struct B : A { int x; }
   struct C : A { int x; }

   int A::*bx = (int(A::*))&B::x;
   int A::*cx = (int(A::*))&C::x;

   bool b1 = (bx == cx); // unspecified
   ```

5. If both refer to (possibly different) members of the same union (11.5), they compare equal.

---

82) As specified in 6.8.3, an object that is not an array element is considered to belong to a single-element array for this purpose.
Otherwise, two pointers to members compare equal if they would refer to the same member of the same most derived object (6.7.2) or the same subobject if indirection with a hypothetical object of the associated class type were performed, otherwise they compare unequal.

**Example 2:**

```cpp
struct B {
    int f();
};
struct L : B {
};
struct R : B {
};
struct D : L, R {
};
int (B::*pb)() = &B::f;
int (L::*pl)() = pb;
int (R::*pr)() = pb;
int (D::*pdl)() = pl;
int (D::*pdr)() = pr;
bool x = (pdl == pdr);  // false
bool y = (pb == pl);  // true
```

— end example

Two operands of type `std::nullptr_t` or one operand of type `std::nullptr_t` and the other a null pointer constant compare equal.

If two operands compare equal, the result is `true` for the `==` operator and `false` for the `!=` operator. If two operands compare unequal, the result is `false` for the `==` operator and `true` for the `!=` operator. Otherwise, the result of each of the operators is unspecified.

If both operands are of arithmetic or enumeration type, the usual arithmetic conversions (7.4) are performed on both operands; each of the operators shall yield `true` if the specified relationship is true and `false` if it is false.

### 7.6.11 Bitwise AND operator

```cpp
and-expression:  
    equality-expression  
    and-expression & equality-expression
```

1 The `&` operator groups left-to-right. The operands shall be of integral or unscoped enumeration type. The usual arithmetic conversions (7.4) are performed. Given the coefficients $x_i$ and $y_i$ of the base-2 representation (6.8.2) of the converted operands $x$ and $y$, the coefficient $r_i$ of the base-2 representation of the result $r$ is 1 if both $x_i$ and $y_i$ are 1, and 0 otherwise.

[Note 1: The result is the bitwise AND function of the operands. — end note]

### 7.6.12 Bitwise exclusive OR operator

```cpp
exclusive-or-expression:  
    and-expression  
    exclusive-or-expression ^ and-expression
```

1 The `^` operator groups left-to-right. The operands shall be of integral or unscoped enumeration type. The usual arithmetic conversions (7.4) are performed. Given the coefficients $x_i$ and $y_i$ of the base-2 representation (6.8.2) of the converted operands $x$ and $y$, the coefficient $r_i$ of the base-2 representation of the result $r$ is 1 if either (but not both) of $x_i$ and $y_i$ are 1, and 0 otherwise.

[Note 1: The result is the bitwise exclusive OR function of the operands. — end note]

### 7.6.13 Bitwise inclusive OR operator

```cpp
inclusive-or-expression:  
    exclusive-or-expression  
    inclusive-or-expression | exclusive-or-expression
```

1 The `|` operator groups left-to-right. The operands shall be of integral or unscoped enumeration type. The usual arithmetic conversions (7.4) are performed. Given the coefficients $x_i$ and $y_i$ of the base-2 representation (6.8.2) of the converted operands $x$ and $y$, the coefficient $r_i$ of the base-2 representation of the result $r$ is 1 if at least one of $x_i$ and $y_i$ are 1, and 0 otherwise.
7.6.14 Logical AND operator  

logical-and-expression:
  inclusive-or-expression
  logical-and-expression && inclusive-or-expression

1 The && operator groups left-to-right. The operands are both contextually converted to bool (7.3). The result is true if both operands are true and false otherwise. Unlike &, && guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is false.

2 The result is a bool. If the second expression is evaluated, the first expression is sequenced before the second expression (6.9.1).

7.6.15 Logical OR operator  

logical-or-expression:
  logical-and-expression
  logical-or-expression || logical-and-expression

1 The || operator groups left-to-right. The operands are both contextually converted to bool (7.3). The result is true if either of its operands is true, and false otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to true.

2 The result is a bool. If the second expression is evaluated, the first expression is sequenced before the second expression (6.9.1).

7.6.16 Conditional operator  

conditional-expression:
  logical-or-expression
  logical-or-expression ? expression : assignment-expression

1 Conditional expressions group right-to-left. The first expression is contextually converted to bool (7.3). It is evaluated and if it is true, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. Only one of the second and third expressions is evaluated. The first expression is sequenced before the second or third expression (6.9.1).

2 If either the second or the third operand has type void, one of the following shall hold:

(2.1) — The second or the third operand (but not both) is a (possibly parenthesized) throw-expression (7.6.18); the result is of the type and value category of the other. The conditional-expression is a bit-field if that operand is a bit-field.

(2.2) — Both the second and the third operands have type void; the result is of type void and is a prvalue.

[Note 1: This includes the case where both operands are throw-expressions. — end note]

3 Otherwise, if the second and third operand are glvalue bit-fields of the same value category and of types cv1 T and cv2 T, respectively, the operands are considered to be of type cv T for the remainder of this subclause, where cv is the union of cv1 and cv2.

4 Otherwise, if the second and third operand have different types and either has (possibly cv-qualified) class type, or if both are glvalues of the same value category and the same type except for cv-qualification, an attempt is made to form an implicit conversion sequence (12.4.4.2) from each of those operands to the type of the other.

[Note 2: Properties such as access, whether an operand is a bit-field, or whether a conversion function is deleted are ignored for that determination. — end note]

Attempts are made to form an implicit conversion sequence from an operand expression E1 of type T1 to a target type related to the type T2 of the operand expression E2 as follows:

(4.1) — If E2 is an lvalue, the target type is “lvalue reference to T2”, but an implicit conversion sequence can only be formed if the reference would bind directly (9.4.4) to a glvalue.

(4.2) — If E2 is an xvalue, the target type is “rvalue reference to T2”, but an implicit conversion sequence can only be formed if the reference would bind directly.

(4.3) — If E2 is a prvalue or if neither of the conversion sequences above can be formed and at least one of the operands has (possibly cv-qualified) class type:
if $T_1$ and $T_2$ are the same class type (ignoring cv-qualification) and $T_2$ is at least as cv-qualified as $T_1$, the target type is $T_2$,

otherwise, if $T_2$ is a base class of $T_1$, the target type is $cv_1T_2$, where $cv_1$ denotes the cv-qualifiers of $T_1$,

otherwise, the target type is the type that $E_2$ would have after applying the lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) standard conversions.

Using this process, it is determined whether an implicit conversion sequence can be formed from the second operand to the target type determined for the third operand, and vice versa. If both sequences can be formed, or one can be formed but it is the ambiguous conversion sequence, the program is ill-formed. If no conversion sequence can be formed, the operands are left unchanged and further checking is performed as described below. Otherwise, if exactly one conversion sequence can be formed, that conversion is applied to the chosen operand and the converted operand is used in place of the original operand for the remainder of this subclause.

[Note 3: The conversion might be ill-formed even if an implicit conversion sequence could be formed. — end note]

If the second and third operands are glvalues of the same value category and have the same type, the result is of that type and value category and it is a bit-field if the second or the third operand is a bit-field, or if both are bit-fields.

Otherwise, the result is a prvalue. If the second and third operands do not have the same type, and either has (possibly cv-qualified) class type, overload resolution is used to determine the conversions (if any) to be applied to the operands (12.4.2.3, 12.7). If the overload resolution fails, the program is ill-formed. Otherwise, the conversions thus determined are applied, and the converted operands are used in place of the original operands for the remainder of this subclause.

Lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) standard conversions are performed on the second and third operands. After those conversions, one of the following shall hold:

1. The second and third operands have the same type; the result is of that type and the result object is initialized using the selected operand.
2. The second and third operands have arithmetic or enumeration type; the usual arithmetic conversions (7.4) are performed to bring them to a common type, and the result is of that type.
3. One or both of the second and third operands have pointer type; pointer conversions (7.3.12), function pointer conversions (7.3.14), and qualification conversions (7.3.6) are performed to bring them to their composite pointer type (7.2.2). The result is of the composite pointer type.
4. One or both of the second and third operands have pointer-to-member type; pointer to member conversions (7.3.13), function pointer conversions (7.3.14), and qualification conversions (7.3.6) are performed to bring them to their composite pointer type (7.2.2). The result is of the composite pointer type.
5. Both the second and third operands have type `std::nullptr_t` or one has that type and the other is a null pointer constant. The result is of type `std::nullptr_t`.

### 7.6.17 Yielding a value

A `yield-expression` shall appear only within a suspension context of a function (7.6.2.4). Let $e$ be the operand of the `yield-expression` and $p$ be an lvalue naming the promise object of the enclosing coroutine (9.5.4), then the `yield-expression` is equivalent to the expression `co_await p.yield_value(e).`

[Example 1]:

```cpp
template <typename T>
struct my_generator {
    struct promise_type {
        T current_value;
        /* ... */
        auto yield_value(T v) {
            current_value = std::move(v);
            return std::suspend_always{};
        }
    }

    void operator()() {
        co_yield current_value;
    }
};
```
my_generator<pair<int,int>> g1() {
    for (int i = 1; i < 10; ++i) co_yield {i,i};
}
my_generator<pair<int,int>> g2() {
    for (int i = 1; i < 10; ++i) co_yield make_pair(i,i);
}
auto f(int x = co_yield 5);
// error: yield-expression outside of function suspension context
int a[] = { co_yield 1 };  // error: yield-expression outside of function suspension context

int main() {
    auto r1 = g1();
    auto r2 = g2();
    assert(std::equal(r1.begin(), r1.end(), r2.begin(), r2.end()));
}
—end example

7.6.18 Throwing an exception
[expr.throw]

throw-expression:
    throw assignment-expression_opt

1 A throw-expression is of type void.

2 Evaluating a throw-expression with an operand throws an exception (14.2); the type of the exception object is determined by removing any top-level cv-qualifiers from the static type of the operand and adjusting the type from “array of $T$” or function type $T$ to “pointer to $T$”.

3 A throw-expression with no operand rethrows the currently handled exception (14.4). The exception is reactivated with the existing exception object; no new exception object is created. The exception is no longer considered to be caught.

[Example 1: An exception handler that cannot completely handle the exception itself can be written like this:
try {
    // ...
} catch (...) {
    // catch all exceptions
    // respond (partially) to exception
    throw;
    // pass the exception to some other handler
}
—end example]

4 If no exception is presently being handled, evaluating a throw-expression with no operand calls std::terminate() (14.6.2).

7.6.19 Assignment and compound assignment operators [expr.ass]

1 The assignment operator (=) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand; their result is an lvalue referring to the left operand. The result in all cases is a bit-field if the left operand is a bit-field. In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression. The right operand is sequenced before the left operand. With respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation.

[Note 1: Therefore, a function call cannot intervene between the lvalue-to-rvalue conversion and the side effect associated with any single compound assignment operator. — end note]
assignment-expression:
  conditional-expression
  yield-expression
  throw-expression
  logical-or-expression
    assignment-operator
    initializer-clause

assignment-operator: one of
  = += /= %= += -= >>= <<= &= ^= |=

In simple assignment (=), the object referred to by the left operand is modified (3.1) by replacing its value with the result of the right operand.

If the right operand is an expression, it is implicitly converted (7.3) to the cv-unqualified type of the left operand.

When the left operand of an assignment operator is a bit-field that cannot represent the value of the expression, the resulting value of the bit-field is implementation-defined.

A simple assignment whose left operand is of a volatile-qualified type is deprecated (D.6) unless the (possibly parenthesized) assignment is a discarded-value expression or an unevaluated operand.

The behavior of an expression of the form \(E_1 \ op = E_2\) is equivalent to \(E_1 = E_1 \ op E_2\) except that \(E_1\) is evaluated only once. Such expressions are deprecated if \(E_1\) has volatile-qualified type; see D.6. For += and -=, \(E_1\) shall either have arithmetic type or be a pointer to a possibly cv-qualified completely-defined object type. In all other cases, \(E_1\) shall have arithmetic type.

If the value being stored in an object is read via another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined.

[Note 2: This restriction applies to the relationship between the left and right sides of the assignment operation; it is not a statement about how the target of the assignment might be aliased in general. See 7.2.1. —end note]

A braced-init-list may appear on the right-hand side of

- an assignment to a scalar, in which case the initializer list shall have at most a single element. The meaning of \(x = \{v\}\), where \(T\) is the scalar type of the expression \(x\), is that of \(x = T\{v\}\). The meaning of \(x = \{\}\) is \(x = T\{\}\).
- an assignment to an object of class type, in which case the initializer list is passed as the argument to the assignment operator function selected by overload resolution (12.6.3.2, 12.4).

[Example 1:

```cpp
complex<double> z;
z = { 1,2 }; // meaning z.operator={1,2}
z += { 1, 2 }; // meaning z.operator+=({1,2})
int a, b;
a = b = { 1 }; // meaning a=b=1;
a = { 1 } = b; // syntax error  
```

—end example]

7.6.20 Comma operator

The comma operator groups left-to-right.

expression:
  assignment-expression
  expression , assignment-expression

A pair of expressions separated by a comma is evaluated left-to-right; the left expression is a discarded-value expression (7.2). The left expression is sequenced before the right expression (6.9.1). The type and value of the result are the type and value of the right operand; the result is of the same value category as its right operand, and is a bit-field if its right operand is a bit-field.

[Note 1: In contexts where the comma token is given special meaning (e.g. function calls (7.6.1.3), lists of initializers (9.4), or template-argument-lists (13.3)), the comma operator as described in this subclause can appear only in parentheses.

[Example 1:

```cpp
f(a, (t=3, t+2), c);
```
has three arguments, the second of which has the value 5. — end example

— end note]

3 [Note 2: A comma expression appearing as the `expr-or-braced-init-list` of a subscripting expression (7.6.1.2) is deprecated; see D.4. — end note]

7.7 Constant expressions

Certain contexts require expressions that satisfy additional requirements as detailed in this subclause; other contexts have different semantics depending on whether or not an expression satisfies these requirements. Expressions that satisfy these requirements, assuming that copy elision (11.10.6) is not performed, are called constant expressions.

[Note 1: Constant expressions can be evaluated during translation. — end note]

constant-expression:

conditional-expression

2 A variable or temporary object o is `constant-initialized` if

(2.1) either it has an initializer or its default-initialization results in some initialization being performed, and

(2.2) the full-expression of its initialization is a constant expression when interpreted as a `constant-expression`, except that if o is an object, that full-expression may also invoke constexpr constructors for o and its subobjects even if those objects are of non-literal class types.

[Note 2: Such a class can have a non-trivial destructor. Within this evaluation, `std::is_constant_evaluated()` (20.15.11) returns true. — end note]

3 A variable is `potentially-constant` if it is constexpr or it has reference or const-qualified integral or enumeration type.

4 A constant-initialized potentially-constant variable V is `usable in constant expressions` at a point P if V’s initializing declaration D is reachable from P and

(4.1) V is constexpr,

(4.2) V is not initialized to a TU-local value, or

(4.3) P is in the same translation unit as D.

An object or reference is `usable in constant expressions` if it is

(4.4) a variable that is usable in constant expressions, or

(4.5) a template parameter object (13.2), or

(4.6) a string literal object (5.13.5), or

(4.7) a temporary object of non-volatile const-qualified literal type whose lifetime is extended (6.7.7) to that of a variable that is usable in constant expressions, or

(4.8) a non-mutable subobject or reference member of any of the above.

5 An expression E is a `core constant expression` unless the evaluation of E, following the rules of the abstract machine (6.9.1), would evaluate one of the following:

(5.1) `this` (7.5.2), except in a constexpr function (9.2.6) that is being evaluated as part of E;

(5.2) an invocation of a non-constexpr function83;

(5.3) an invocation of an undefined constexpr function;

(5.4) an invocation of an instantiated constexpr function that fails to satisfy the requirements for a constexpr function;

(5.5) an invocation of a virtual function (11.7.3) for an object unless

(5.5.1) the object is usable in constant expressions or

(5.5.2) its lifetime began within the evaluation of E;

(5.6) an expression that would exceed the implementation-defined limits (see Annex B);

83) Overload resolution (12.4) is applied as usual.
— an operation that would have undefined behavior as specified in Clause 4 through Clause 15\(^{84}\);

— an lvalue-to-rvalue conversion (7.3.2) unless it is applied to

— a non-volatile glvalue that refers to an object that is usable in constant expressions, or

— a non-volatile glvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of \(E\);

— an lvalue-to-rvalue conversion that is applied to a glvalue that refers to a non-active member of a union or a subobject thereof;

— an lvalue-to-rvalue conversion that is applied to an object with an indeterminate value (6.7.4);

— an invocation of an implicitly-defined copy/move constructor or copy/move assignment operator for a union whose active member (if any) is mutable, unless the lifetime of the union object began within the evaluation of \(E\);

— an \textit{id-expression} that refers to a variable or data member of reference type unless the reference has a preceding initialization and either

— it is usable in constant expressions or

— its lifetime began within the evaluation of \(E\);

— in a \textit{lambda-expression}, a reference to \texttt{this} or to a variable with automatic storage duration defined outside that \textit{lambda-expression}, where the reference would be an odr-use (6.3, 7.5.5);

\[\text{Example 1:}\]

```cpp
void g() {
    const int n = 0;
    [=] {
        constexpr int i = n;  // OK, \(n\) is not odr-used here
        constexpr int j = *\&n;  // error: \&n would be an odr-use of \(n\)
    };
}
\]

\[\text{end example}\]

\[\text{Note 3:}\] If the odr-use occurs in an invocation of a function call operator of a closure type, it no longer refers to \texttt{this} or to an enclosing automatic variable due to the transformation (7.5.5.3) of the \textit{id-expression} into an access of the corresponding data member.

\[\text{Example 2:}\]

```cpp
auto monad = [] (auto v) { return [=] { return v; }();
    auto bind = [] (auto m) {
        return [=] (auto fvm) { return fvm(m()); };
    };
    // OK to capture objects with automatic storage duration created during constant expression evaluation.
    static_assert(bind(monomad(2)) (monad)() == monad(2)());
}
\]

\[\text{end example}\]

\[\text{end note}\]

— a conversion from type \(cv\) \texttt{void*\*} to a pointer-to-object type;

— a \texttt{reinterpret_cast} (7.6.1.10);

— a modification of an object (7.6.19, 7.6.1.6, 7.6.2.3) unless it is applied to a non-volatile lvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of \(E\);

— a \textit{new-expression} (7.6.2.8), unless the selected allocation function is a replaceable global allocation function (17.6.3.2, 17.6.3.3) and the allocated storage is deallocated within the evaluation of \(E\);

— a \textit{delete-expression} (7.6.2.9), unless it deallocates a region of storage allocated within the evaluation of \(E\);

— a call to an instance of \texttt{std::allocator}\textless\texttt{T}\texttt{::allocate} (20.10.10.2), unless the allocated storage is deallocated within the evaluation of \(E\);

\[\text{\(84\)}\] This includes, for example, signed integer overflow (7.2), certain pointer arithmetic (7.6.6), division by zero (7.6.5), or certain shift operations (7.6.7).
— a call to an instance of `std::allocator<T>::deallocate` (20.10.10.2), unless it deallocates a region of storage allocated within the evaluation of \( E \);

— an `await-expression` (7.6.2.4);

— a `yield-expression` (7.6.17);

— a three-way comparison (7.6.8), relational (7.6.9), or equality (7.6.10) operator where the result is unspecified;

— a `throw-expression` (7.6.18);

— a `dynamic_cast` (7.6.1.7) or `typeid` (7.6.1.8) expression that would throw an exception;

— an `asm-declaration` (9.10); or

— an invocation of the `va_arg` macro (17.13.2).

If \( E \) satisfies the constraints of a core constant expression, but evaluation of \( E \) would evaluate an operation that has undefined behavior as specified in Clause 16 through Clause 32, or an invocation of the `va_start` macro (17.13.2), it is unspecified whether \( E \) is a core constant expression.

**Example 3:**

```cpp
int x;    // not constant
struct A {
    constexpr A(bool b) : m(b?42:x) {} 
    int m;
};
constexpr int v = A(true).m;    // OK: constructor call initializes m with the value 42
constexpr int w = A(false).m;   // error: initializer for m is x, which is non-constant

constexpr int f1(int k) {
    constexpr int x = k;    // error: x is not initialized by a constant expression
    // because lifetime of k began outside the initializer of x
    return x;
}
constexpr int f2(int k) {    // OK: not required to be a constant expression
    int x = k;
    // because x is not constexpr
    return x;
}

constexpr int incr(int &n) {    // OK: not required to be a core constant expression
    return ++n;
}
constexpr int g(int k) {       // error: incr(k) is not a core constant expression
    constexpr int x = incr(k);    // because lifetime of k began outside the expression incr(k)
    return x;
}
constexpr int h(int k) {       // OK: incr(k) is not required to be a core constant expression
    int x = incr(k);
    return x;
}
constexpr int y = h(1);        // OK: initializes y with the value 2
    // h(1) is a core constant expression because
    // the lifetime of k begins inside h(1)

— end example
```

6 For the purposes of determining whether an expression \( E \) is a core constant expression, the evaluation of a call to a member function of `std::allocator<T>` as defined in 20.10.10.2, where \( T \) is a literal type, does not disqualify \( E \) from being a core constant expression, even if the actual evaluation of such a call would otherwise fail the requirements for a core constant expression. Similarly, the evaluation of a call to `std::destroy_at`, `std::ranges::destroy_at`, `std::construct_at`, or `std::ranges::construct_at` does not disqualify \( E \) from being a core constant expression unless:
(6.1) — for a call to `std::construct_at` or `std::ranges::construct_at`, the first argument, of type `T*`, does not point to storage allocated with `std::allocator<T>` or to an object whose lifetime began within the evaluation of `E`, or the evaluation of the underlying constructor call disqualifies `E` from being a core constant expression, or

(6.2) — for a call to `std::destroy_at` or `std::ranges::destroy_at`, the first argument, of type `T*`, does not point to storage allocated with `std::allocator<T>` or to an object whose lifetime began within the evaluation of `E`, or the evaluation of the underlying destructor call disqualifies `E` from being a core constant expression.

An object `a` is said to have `constant destruction` if:

(7.1) — it is not of class type nor (possibly multi-dimensional) array thereof, or

(7.2) — it is of class type or (possibly multi-dimensional) array thereof, that class type has a `constexpr` destructor, and for a hypothetical expression `E` whose only effect is to destroy `a`, `E` would be a core constant expression if the lifetime of `a` and its non-mutable subobjects (but not its mutable subobjects) were considered to start within `E`.

An `integral constant expression` is an expression of integral or unscoped enumeration type, implicitly converted to a prvalue, where the converted expression is a core constant expression.

[Note 4: Such expressions can be used as bit-field lengths (11.4.10), as enumerator initializers if the underlying type is not fixed (9.7.1), and as alignments (9.12.2). — end note]

If an expression of literal class type is used in a context where an integral constant expression is required, then that expression is contextually implicitly converted (7.3) to an integral or unscoped enumeration type and the selected conversion function shall be `constexpr`.

[Example 4:

```cpp
struct A {
    constexpr A(int i) : val(i) { }  
    constexpr operator int() const { return val; }
    constexpr operator long() const { return 42; }
    private:
        int val;
    private:
    constexpr A a = alignof(int);
    alignas(a) int n;  // error: ambiguous conversion
    struct B { int n : a; };  // error: ambiguous conversion
};
```

— end example]

A `converted constant expression` of type `T` is an expression, implicitly converted to type `T`, where the converted expression is a constant expression and the implicit conversion sequence contains only

(10.1) — user-defined conversions,
(10.2) — lvalue-to-rvalue conversions (7.3.2),
(10.3) — array-to-pointer conversions (7.3.3),
(10.4) — function-to-pointer conversions (7.3.4),
(10.5) — qualification conversions (7.3.6),
(10.6) — integral promotions (7.3.7),
(10.7) — integral conversions (7.3.9) other than narrowing conversions (9.4.5),
(10.8) — null pointer conversions (7.3.12) from `std::nullptr_t`,
(10.9) — null member pointer conversions (7.3.13) from `std::nullptr_t`, and
(10.10) — function pointer conversions (7.3.14),

and where the reference binding (if any) binds directly.

[Note 5: Such expressions can be used in `new` expressions (7.6.2.8), as case expressions (8.5.3), as enumerator initializers if the underlying type is fixed (9.7.1), as array bounds (9.3.4.5), and as non-type template arguments (13.4). — end note]
A contextually converted constant expression of type bool is an expression, contextually converted to bool (7.3), where the converted expression is a constant expression and the conversion sequence contains only the conversions above.

A constant expression is either a glvalue core constant expression that refers to an entity that is a permitted result of a constant expression (as defined below), or a prvalue core constant expression whose value satisfies the following constraints:

11. A constant expression is either a glvalue core constant expression that refers to an entity that is a permitted result of a constant expression (as defined below), or a prvalue core constant expression whose value satisfies the following constraints:

11.1 — if the value is an object of class type, each non-static data member of reference type refers to an entity that is a permitted result of a constant expression,

11.2 — if the value is of pointer type, it contains the address of an object with static storage duration, the address past the end of such an object (7.6.6), the address of a non-immediate function, or a null pointer value,

11.3 — if the value is of pointer-to-member-function type, it does not designate an immediate function, and

11.4 — if the value is an object of class or array type, each subobject satisfies these constraints for the value.

An entity is a permitted result of a constant expression if it is an object with static storage duration that either is not a temporary object or is a temporary object whose value satisfies the above constraints, or if it is a non-immediate function.

[Example 5:]
```c
consteval int f() { return 42; }
consteval auto g() { return f; }
consteval int h(int (*p)() = g()) { return p(); }
constexpr int r = h(); // OK
constexpr auto e = g(); // error: a pointer to an immediate function is not a permitted result of a constant expression
```

12. Recommended practice: Implementations should provide consistent results of floating-point evaluations, irrespective of whether the evaluation is performed during translation or during program execution.

[Note 6: Since this document imposes no restrictions on the accuracy of floating-point operations, it is unspecified whether the evaluation of a floating-point expression during translation yields the same result as the evaluation of the same expression (or the same operations on the same values) during program execution.]

[Example 6:]
```c
bool f() {
    char array[1 + int(1 + 0.2 - 0.1 - 0.1)]; // Must be evaluated during translation
    int size = 1 + int(1 + 0.2 - 0.1 - 0.1); // May be evaluated at runtime
    return sizeof(array) == size;
}
```

It is unspecified whether the value of f() will be true or false. —end example]

13. An expression or conversion is in an immediate function context if it is potentially evaluated and its innermost non-block scope is a function parameter scope of an immediate function. An expression or conversion is an immediate invocation if it is a potentially-evaluated explicit or implicit invocation of an immediate function and is not in an immediate function context. An immediate invocation shall be a constant expression.

14. An expression or conversion is manifestly constant-evaluated if it is:

14.1 — a constant-expression, or
14.2 — the condition of a constexpr if statement (8.5.2), or
14.3 — an immediate invocation, or
14.4 — the result of substitution into an atomic constraint expression to determine whether it is satisfied (13.5.2.3), or
14.5 — the initializer of a variable that is usable in constant expressions or has constant initialization (6.9.3.2).85

[Example 7:]
```c
template<bool> struct X {};
```

85 Testing this condition might involve a trial evaluation of its initializer as described above.
\texttt{X<\texttt{std::is\_constant\_evaluated}()> x;} \quad // \text{type \texttt{X<true>}}

\texttt{int y;} \quad // \text{type \texttt{int}}

\texttt{const int a = \texttt{std::is\_constant\_evaluated}() ? y : 1;} \quad // \text{dynamic initialization to 1}
\texttt{double z[a];} \quad // \text{error: a is not usable}

\texttt{const int b = \texttt{std::is\_constant\_evaluated}() ? 2 : y;} \quad // \text{static initialization to 2}
\texttt{int c = y + (\texttt{std::is\_constant\_evaluated}() ? 2 : y);} \quad // \text{dynamic initialization to} \texttt{y+y}

\texttt{constexpr int f() \{ \}
\texttt{const int n = \texttt{std::is\_constant\_evaluated}() ? 13 : 17;} \quad // \texttt{n is 13}
\texttt{int m = \texttt{std::is\_constant\_evaluated}() ? 13 : 17;} \quad // \texttt{m might be 13 or 17 (see below)}
\texttt{char arr[n] = \{}; \quad // \texttt{char[13]}
\texttt{return m + \texttt{sizeof(arr)};} \quad // \texttt{m + sizeof(arr)}
\texttt{\} \quad // \texttt{m is 13; initialized to 26}
\texttt{int p = f();} \quad // \texttt{m is 17 for this call; initialized to 56}
\texttt{int q = p + f();} \quad // \texttt{m is 17 for this call; initialized to 56}
\texttt{\}} \quad // \texttt{m is 17 for this call; initialized to 56}

[\textbf{Note 7}: A manifestly constant-evaluated expression is evaluated even in an unevaluated operand. — end note]

An expression or conversion is \textbf{potentially constant evaluated} if it is:
\begin{itemize}
\item \textbf{(15.1)} a manifestly constant-evaluated expression,
\item \textbf{(15.2)} a potentially-evaluated expression (6.3),
\item \textbf{(15.3)} an immediate subexpression of a \textit{braced-init-list},\textsuperscript{86}
\item \textbf{(15.4)} an expression of the form \texttt{& cast-expression} that occurs within a templated entity,\textsuperscript{87} or
\item \textbf{(15.5)} a subexpression of one of the above that is not a subexpression of a nested unevaluated operand.
\end{itemize}

A function or variable is \textbf{needed for constant evaluation} if it is:
\begin{itemize}
\item \textbf{(15.6)} a \texttt{constexpr} function that is named by an expression (6.3) that is potentially constant evaluated, or
\item \textbf{(15.7)} a variable whose name appears as a potentially constant evaluated expression that is either a \texttt{constexpr} variable or is of non-volatile const-qualified integral type or of reference type.
\end{itemize}

\textsuperscript{86} Constant evaluation might be necessary to determine whether a narrowing conversion is performed (9.4.5).

\textsuperscript{87} Constant evaluation might be necessary to determine whether such an expression is value-dependent (13.8.3.4).
8 Statements

8.1 Preamble

Except as indicated, statements are executed in sequence.

statement:
  labeled-statement
  attribute-specifier-seq? expression-statement
  attribute-specifier-seq? compound-statement
  attribute-specifier-seq? selection-statement
  attribute-specifier-seq? iteration-statement
  attribute-specifier-seq? jump-statement
  declaration-statement
  attribute-specifier-seq? try-block

init-statement:
  expression-statement
  simple-declaration

condition:
  expression
  attribute-specifier-seq? decl-specifier-seq declarator brace-or-equal-initializer

The optional attribute-specifier-seq appertains to the respective statement.

2 A substatement of a statement is one of the following:
   (2.1) for a labeled-statement, its contained statement,
   (2.2) for a compound-statement, any statement of its statement-seq,
   (2.3) for a selection-statement, any of its statements (but not its init-statement), or
   (2.4) for an iteration-statement, its contained statement (but not an init-statement).
[Note 1: The compound-statement of a lambda-expression is not a substatement of the statement (if any) in which the lambda-expression lexically appears. — end note]

3 A statement $S_1$ encloses a statement $S_2$ if
   (3.1) $S_2$ is a substatement of $S_1$ (Clause 9),
   (3.2) $S_1$ is a selection-statement or iteration-statement and $S_2$ is the init-statement of $S_1$,
   (3.3) $S_1$ is a try-block and $S_2$ is its compound-statement or any of the compound-statements of its handlers, or
   (3.4) $S_1$ encloses a statement $S_3$ and $S_3$ encloses $S_2$.

4 The rules for conditions apply both to selection-statements and to the for and while statements (8.6). A condition that is not an expression is a declaration (Clause 9). The declarator shall not specify a function or an array. The decl-specifier-seq shall not define a class or enumeration. If the auto type-specifier appears in the decl-specifier-seq, the type of the identifier being declared is deduced from the initializer as described in 9.2.9.6.
[Note 2: A name introduced in a selection-statement or iteration-statement outside of any substatement is in scope from its point of declaration until the end of the statement’s substatements. Such a name cannot be redeclared in the outermost block of any of the substatements (6.4.3). — end note]

6 The value of a condition that is an initialized declaration in a statement other than a switch statement is the value of the declared variable contextually converted to bool (7.3). If that conversion is ill-formed, the program is ill-formed. The value of a condition that is an initialized declaration in a switch statement is the value of the declared variable if it has integral or enumeration type, or of that variable implicitly converted to integral or enumeration type otherwise. The value of a condition that is an expression is the value of the expression, contextually converted to bool for statements other than switch; if that conversion is ill-formed, the program is ill-formed. The value of the condition will be referred to as simply “the condition” where the usage is unambiguous.

7 If a condition can be syntactically resolved as either an expression or the declaration of a block-scope name, it is interpreted as a declaration:

§ 8.1
8 In the `decl-specifier-seq` of a `condition`, each `decl-specifier` shall be either a `type-specifier` or `constexpr`.

### 8.2 Labeled statement

1 A statement can be labeled.

```
labeled-statement:
  attribute-specifier-seq_opt identifier : statement
  attribute-specifier-seq_opt case constant-expression : statement
  attribute-specifier-seq_opt default : statement
```

The optional `attribute-specifier-seq` appertains to the label. An `identifier label` declares the identifier. The only use of an identifier label is as the target of a `goto`. The scope of a label is the function in which it appears. Labels shall not be redeclared within a function. A label can be used in a `goto` statement before its declaration. Labels have their own name space and do not interfere with other identifiers.

[Note 1: A label can have the same name as another declaration in the same scope or a `template-parameter` from an enclosing scope. Unqualified name lookup (6.5.2) ignores labels. — end note]

2 Case labels and default labels shall occur only in `switch` statements.

### 8.3 Expression statement

1 Expression statements have the form

```
expression-statement:
  expression_opt ;
```

The expression is a discarded-value expression (7.2.3). All side effects from an expression statement are completed before the next statement is executed. An expression statement with the expression missing is called a `null statement`.

[Note 1: Most statements are expression statements — usually assignments or function calls. A null statement is useful to carry a label just before the `}` of a compound statement and to supply a null body to an iteration statement such as a `while` statement (8.6.2). — end note]

### 8.4 Compound statement or block

1 A `compound statement` (also known as a block) groups a sequence of statements into a single statement.

```
compound-statement:
  { statement-seq_opt }

statement-seq:
  statement
  statement-seq statement
```

A compound statement defines a block scope (6.4).

[Note 1: A declaration is a `statement` (8.8). — end note]

### 8.5 Selection statements

#### 8.5.1 General

1 Selection statements choose one of several flows of control.

```
selection-statement:
  if constexpr_opt ( init-statement_opt condition ) statement
  if constexpr_opt ( init-statement_opt condition ) statement else statement
  switch ( init-statement_opt condition ) statement
```

See 9.3.4 for the optional `attribute-specifier-seq` in a condition.

[Note 1: An `init-statement` ends with a semicolon. — end note]

2 The substatement in a `selection-statement` (each substatement, in the `else` form of the `if` statement) implicitly defines a block scope (6.4). If the substatement in a `selection-statement` is a single statement and not a `compound-statement`, it is as if it was rewritten to be a `compound-statement` containing the original substatement.

[Example 1:
```
if (x)
  int i;
```]
can be equivalently rewritten as

```c
if (x) {
    int i;
}
```

Thus after the `if` statement, `i` is no longer in scope. —end example]

### 8.5.2 The if statement

1 If the condition (8.5) yields `true` the first substatement is executed. If the `else` part of the selection statement is present and the condition yields `false`, the second substatement is executed. If the first substatement is reached via a label, the condition is not evaluated and the second substatement is not executed. In the second form of `if` statement (the one including `else`), if the first substatement is also an `if` statement then that inner `if` statement shall contain an `else` part.88

2 If the `if` statement is of the form `if constexpr`, the value of the condition shall be a contextually converted constant expression of type `bool (7.7)`; this form is called a `constexpr if` statement. If the value of the converted condition is `false`, the first substatement is a discarded statement, otherwise the second substatement, if present, is a discarded statement. During the instantiation of an enclosing templated entity (13.1), if the condition is not value-dependent after its instantiation, the discarded substatement (if any) is not instantiated.

[Note 1: Odr-uses (6.3) in a discarded statement do not require an entity to be defined. —end note]

A `case` or `default` label appearing within such an `if` statement shall be associated with a `switch` statement (8.5.3) within the same `if` statement. A label (8.2) declared in a substatement of a constexpr if statement shall only be referred to by a statement (8.7.6) in the same substatement.

[Example 1]

```c
template<typename T, typename ... Rest> void g(T&& p, Rest&& ...rs) {
    // ... handle p
    if constexpr (sizeof...(rs) > 0)
        g(rs...); // never instantiated with an empty argument list
}
extern int x; // no definition of x required
int f() {
    if constexpr (true)
        return 0;
    else if (x)
        return x;
    else
        return -x;
}
—end example]

3 An `if` statement of the form

```c
if constexpr opt ( init-statement condition ) statement
```

is equivalent to

```c
{
    init-statement
    if constexpr opt ( condition ) statement
}
```

and an `if` statement of the form

```c
if constexpr opt ( init-statement condition ) statement else statement
```

is equivalent to

```c
{
    init-statement
    if constexpr opt ( condition ) statement else statement
}
```

88) In other words, the `else` is associated with the nearest un-elsed `if`. 
except that names declared in the init-statement are in the same declarative region as those declared in the condition.

8.5.3 The switch statement [stmt.switch]

1 The switch statement causes control to be transferred to one of several statements depending on the value of a condition.

2 The condition shall be of integral type, enumeration type, or class type. If of class type, the condition is contextually implicitly converted (7.3) to an integral or enumeration type. If the (possibly converted) type is subject to integral promotions (7.3.7), the condition is converted to the promoted type. Any statement within the switch statement can be labeled with one or more case labels as follows:

   case constant-expression :

where the constant-expression shall be a converted constant expression (7.7) of the adjusted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion.

3 There shall be at most one label of the form

   default :

within a switch statement.

4 Switch statements can be nested; a case or default label is associated with the smallest switch enclosing it.

5 When the switch statement is executed, its condition is evaluated. If one of the case constants has the same value as the condition, control is passed to the statement following the matched case label. If no case constant matches the condition, and if there is a default label, control passes to the statement labeled by the default label. If no case matches and if there is no default then none of the statements in the switch is executed.

6 case and default labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see break, 8.7.2.

[Note 1: Usually, the substatement that is the subject of a switch is compound and case and default labels appear on the top-level statements contained within the (compound) substatement, but this is not required. Declarations can appear in the substatement of a switch statement. — end note]

7 A switch statement of the form

   switch ( init-statement condition ) statement

is equivalent to

   {
   init-statement
   switch ( condition ) statement
   }

except that names declared in the init-statement are in the same declarative region as those declared in the condition.

8.6 Iteration statements [stmt.iter]

8.6.1 General [stmt.iter.general]

1 Iteration statements specify looping.

   iteration-statement:
   while ( condition ) statement
   do statement while ( expression ) ;
   for ( init-statement condition_opt ; expression_opt ) statement
   for ( init-statement,opt for-range-declaration : for-range-initializer ) statement

   for-range-declaration:
   attribute-specifier-seq_opt decl-specifier-seq declarator
   attribute-specifier-seq_opt decl-specifier-seq ref-qualifier_opt [ identifier-list ]

   for-range-initializer:
   expr-or-braced-init-list

See 9.3.4 for the optional attribute-specifier-seq in a for-range-declaration.

[Note 1: An init-statement ends with a semicolon. — end note]
The substatement in an *iteration-statement* implicitly defines a block scope (6.4) which is entered and exited each time through the loop. If the substatement in an *iteration-statement* is a single statement and not a *compound-statement*, it is as if it was rewritten to be a *compound-statement* containing the original statement.

*Example 1:*

```c
while (--x >= 0)
    int i;
```

can be equivalently rewritten as

```c
while (--x >= 0) {
    int i;
}
```

Thus after the *while* statement, `i` is no longer in scope. —end example*

If a name introduced in an *init-statement* or *for-range-declaration* is redeclared in the outermost block of the substatement, the program is ill-formed.

*Example 2:*

```c
void f() {
    for (int i = 0; i < 10; ++i)
        int i = 0; // error: redeclaration
    for (int i : { 1, 2, 3 })
        int i = 1; // error: redeclaration
}
```

—end example*

### 8.6.2 The *while* statement

*In the *while* statement the substatement is executed repeatedly until the value of the condition (8.5) becomes false. The test takes place before each execution of the substatement.*

*When the condition of a *while* statement is a declaration, the scope of the variable that is declared extends from its point of declaration (6.4.2) to the end of the *while* statement. A *while* statement is equivalent to* 

```
label :
{
    if (condition) {
        statement
        goto label ;
    }
}
```

*Note 1: The variable created in the condition is destroyed and created with each iteration of the loop.  
*Example 1:*

```c
struct A {
    int val;
    A(int i) : val(i) { }
    ~A() {}
    operator bool() { return val != 0; }
};
int i = 1;
while (A a = i) { // ...
    i = 0;
}
```

In the while-loop, the constructor and destructor are each called twice, once for the condition that succeeds and once for the condition that fails. —end example*

—end note*

### 8.6.3 The *do* statement

*The expression is contextually converted to *bool* (7.3); if that conversion is ill-formed, the program is ill-formed.*
In the do statement the substatement is executed repeatedly until the value of the expression becomes false. The test takes place after each execution of the statement.

8.6.4 The for statement

The for statement

```
for ( init-statement condition_opt ; expression_opt ) statement
```

is equivalent to

```
{ 
  init-statement
  while ( condition ) {
    statement
    expression ;
  }
}
```

except that names declared in the init-statement are in the same declarative region as those declared in the condition, and except that a continue in statement (not enclosed in another iteration statement) will execute expression before re-evaluating condition.

[Note 1: Thus the first statement specifies initialization for the loop; the condition (8.5) specifies a test, sequenced before each iteration, such that the loop is exited when the condition becomes false; the expression often specifies incrementing that is sequenced after each iteration. — end note]

Either or both of the condition and the expression can be omitted. A missing condition makes the implied while clause equivalent to while(true).

If the init-statement is a declaration, the scope of the name(s) declared extends to the end of the for statement.

[Example 1:

```
int i = 42;
int a[10];

for (int i = 0; i < 10; i++)
  a[i] = i;

int j = i; // j = 42
```

— end example]

8.6.5 The range-based for statement

The range-based for statement

```
for ( init-statement_opt for-range-declaration : for-range-initializer ) statement
```

is equivalent to

```
{ 
  init-statement_opt
  auto &range = for-range-initializer ;
  auto begin = begin-expr ;
  auto end = end-expr ;
  for ( ; begin != end ; ++begin ) {
    for-range-declaration = * begin ;
    statement
  }
}
```

where

(1.1) — if the for-range-initializer is an expression, it is regarded as if it were surrounded by parentheses (so that a comma operator cannot be reinterpreted as delimiting two init-declarators);

(1.2) — range, begin, and end are variables defined for exposition only; and

(1.3) — begin-expr and end-expr are determined as follows:
— if the `for-range-initializer` is an expression of array type `R`, `begin-expr` and `end-expr` are `range` and `range + N`, respectively, where `N` is the array bound. If `R` is an array of unknown bound or an array of incomplete type, the program is ill-formed;

— if the `for-range-initializer` is an expression of class type `C`, the `unqualified-ids` `begin` and `end` are looked up in the scope of `C` as if by class member access lookup (6.5.6), and if both find at least one declaration, `begin-expr` and `end-expr` are `range . begin()` and `range . end()`, respectively;

— otherwise, `begin-expr` and `end-expr` are `begin(range)` and `end(range)`, respectively, where `begin` and `end` are looked up in the associated namespaces (6.5.3).

[Note 1: Ordinary unqualified lookup (6.5.2) is not performed. — end note]

### Example 1:
```c
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
   x *= 2;
```

— end example

2 In the `decl-specifier-seq` of a `for-range-declaration`, each `decl-specifier` shall be either a `type-specifier` or `constexpr`. The `decl-specifier-seq` shall not define a class or enumeration.

#### 8.7 Jump statements

##### 8.7.1 General

Jump statements unconditionally transfer control.

```
jump-statement:
   break ;
   continue ;
   return expr-or-braced-init-listopt ;
   coroutine-return-statement
   goto identifier ;
```

2 On exit from a scope (however accomplished), objects with automatic storage duration (6.7.5.4) that have been constructed in that scope are destroyed in the reverse order of their construction.

[Note 1: For temporaries, see 6.7.7. — end note]

Transfer out of a loop, out of a block, or back past an initialized variable with automatic storage duration involves the destruction of objects with automatic storage duration that are in scope at the point transferred from but not at the point transferred to. (See 8.8 for transfers into blocks).

[Note 2: However, the program can be terminated (by calling `std::exit()` or `std::abort()` (17.5), for example) without destroying objects with automatic storage duration. — end note]

[Note 3: A suspension of a coroutine (7.6.2.4) is not considered to be an exit from a scope. — end note]

##### 8.7.2 The break statement

1 The `break` statement shall occur only in an `iteration-statement` or a `switch` statement and causes termination of the smallest enclosing `iteration-statement` or `switch` statement; control passes to the statement following the terminated statement, if any.

##### 8.7.3 The continue statement

1 The `continue` statement shall occur only in an `iteration-statement` and causes control to pass to the loop-continuation portion of the smallest enclosing `iteration-statement`, that is, to the end of the loop. More precisely, in each of the statements

```c
while (foo) {
   do {
      { // ...
         { // ...
            { // ...
               contin: ;
            } contin: ;
         } contin: ;
      } while (foo);
   }
}
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto contin`. 

§ 8.7.3
8.7.4 The return statement

A function returns to its caller by the return statement. The expr-or-braced-init-list of a return statement is called its operand. A return statement with no operand shall be used only in a function whose return type is cv void, a constructor (11.4.5), or a destructor (11.4.7). A return statement with an operand of type void shall be used only in a function whose return type is cv void. A return statement with any other operand shall be used only in a function whose return type is not cv void; the return statement initializes the glvalue result or prvalue result object of the (explicit or implicit) function call by copy-initialization (9.4) from the operand. 

[Note 1: A return statement can involve an invocation of a constructor to perform a copy or move of the operand if it is not a prvalue or if its type differs from the return type of the function. A copy operation associated with a return statement can be elided or converted to a move operation if an automatic storage duration variable is returned (11.10.6). —end note]

[Example 1:  
std::pair<std::string,int> f(const char* p, int x) {  
    return {p,x};  
}  
—end example]

The destructor for the result object is potentially invoked (11.4.7, 14.3).

[Example 2:  
class A {  
    ~A() {}  
};  
A f() { return A(); }  
// error: destructor of A is private (even though it is never invoked)  
—end example]

Flowing off the end of a constructor, a destructor, or a non-coroutine function with a cv void return type is equivalent to a return with no operand. Otherwise, flowing off the end of a function other than main (6.9.3.1) or a coroutine (9.5.4) results in undefined behavior.

8.7.5 The co_return statement

A coroutine returns to its caller or resumer (9.5.4) by the co_return statement or when suspended (7.6.2.4). A coroutine shall not enclose a return statement (8.7.4).

[Note 1: For this determination, it is irrelevant whether the return statement is enclosed by a discarded statement (8.5.2). —end note]

The expr-or-braced-init-list of a co_return statement is called its operand. Let p be an lvalue naming the coroutine promise object (9.5.4). A co_return statement is equivalent to:

{ S; goto final-suspend; }

where final-suspend is the exposition-only label defined in 9.5.4 and S is defined as follows:

(2.1) If the operand is a braced-init-list or an expression of non-void type, S is p.return_value(expr-or-braced-init-list). The expression S shall be a prvalue of type void.

(2.2) Otherwise, S is the compound-statement { expression_opt; p.return_void(); }. The expression p.return_void() shall be a prvalue of type void.

If p.return_void() is a valid expression, flowing off the end of a coroutine is equivalent to a co_return with no operand; otherwise flowing off the end of a coroutine results in undefined behavior.

8.7.6 The goto statement

The goto statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (8.2) located in the current function.
8.8 Declaration statement

A declaration statement introduces one or more new identifiers into a block; it has the form

```
declaration-statement:
  block-declaration
```

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

Variables with automatic storage duration (6.7.5.4) are initialized each time their declaration-statement is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (8.7).

It is possible to transfer into a block, but not in a way that bypasses declarations with initialization (including ones in conditions and init-statements). A program that jumps from a point where a variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has vacuous initialization (6.7.3). In such a case, the variables with vacuous initialization are constructed in the order of their declaration.

```
[Example 1:
  void f() {
    // ...
    goto lx;         // error: jump into scope of a
    // ...
    ly:
    X a = 1;
    // ...
    lx:
    goto ly;        // OK, jump implies destructor call for a followed by
                    // construction again immediately following label ly
  }
  —end example]
```

Dynamic initialization of a block-scope variable with static storage duration (6.7.5.2) or thread storage duration (6.7.5.3) is performed the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.

```
[Note 1: A conforming implementation cannot introduce any deadlock around execution of the initializer. Deadlocks might still be caused by the program logic; the implementation need only avoid deadlocks due to its own synchronization operations. —end note]
```

If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined.

```
[Example 2:
  int foo(int i) {
    static int s = foo(2*i);     // undefined behavior: recursive call
    return i+1;
  }
  —end example]
```

A block-scope object with static or thread storage duration will be destroyed if and only if it was constructed.

```
[Note 2: 6.9.3.4 describes the order in which block-scope objects with static and thread storage duration are destroyed. —end note]
```

8.9 Ambiguity resolution

There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion (7.6.1.4) as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a ( . In those cases the statement is a declaration.

---

89) The transfer from the condition of a switch statement to a case label is considered a jump in this respect.
2 [Note 1: If the statement cannot syntactically be a declaration, there is no ambiguity, so this rule does not apply. The whole statement might need to be examined to determine whether this is the case. This resolves the meaning of many examples.]

[Example 1: Assuming T is a simple-type-specifier (9.2.9),]

```
T(a)->m = 7; // expression-statement
T(a)++; // expression-statement
T(a,5)<<c; // expression-statement

T(*d)(int); // declaration
T(e)[5]; // declaration
T(f) = { 1, 2 }; // declaration
T(*g)(double(3)); // declaration
```

In the last example above, g, which is a pointer to T, is initialized to double(3). This is of course ill-formed for semantic reasons, but that does not affect the syntactic analysis. —end example]

The remaining cases are declarations.

[Example 2:

```
class T {
    // ...
    public:
    T();
    T(int);
    T(int, int);
};
T(a); // declaration
T(*b)(); // declaration
T(c)=7; // declaration
T(d),e,f=3; // declaration
extern int h;
T(g)(h,2); // declaration
```

—end example]

—end note]

3 The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are type-names or not, is not generally used in or changed by the disambiguation. Class templates are instantiated as necessary to determine if a qualified name is a type-name. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required.

[Note 2: This can occur only when the name is declared earlier in the declaration. —end note]

[Example 3:

```
struct T1 {
    T1 operator()(int x) { return T1(x); }
    int operator=(int x) { return x; }
    T1(int) {} 
};
struct T2 { T2(int){ } }; 
int a, (*(*b)(T2藠)(int), c, d;

void f() {
    // disambiguation requires this to be parsed as a declaration:
    T1(a) = 3,
    T2(4),
    (***b)(T2(c)))(int(d)); // T2 will be declared as a variable of type T1, but this will not
    // allow the last part of the declaration to parse properly,
    // since it depends on T2 being a type-name
}
```

—end example]
9 Declarations

9.1 Preamble

Declarations generally specify how names are to be interpreted. Declarations have the form

\[
\text{declaration-seq}:
\text{declaration }
\text{declaration-seq declaration}
\]

declaration:
  \[
  \text{block-declaration}
  \text{nodeclspec-function-declaration}
  \text{function-definition}
  \text{template-declaration}
  \text{deduction-guide}
  \text{explicit-instantiation}
  \text{explicit-specialization}
  \text{export-declaration}
  \text{linkage-specification}
  \text{namespace-definition}
  \text{empty-declaration}
  \text{attribute-declaration}
  \text{module-import-declaration}
\]

block-declaration:
  \[
  \text{simple-declaration}
  \text{asm-declaration}
  \text{namespace-alias-definition}
  \text{using-declaration}
  \text{using-enum-declaration}
  \text{using-directive}
  \text{static_assert-declaration}
  \text{alias-declaration}
  \text{opaque-enum-declaration}
\]

nodeclspec-function-declaration:
  \[
  \text{attribute-specifier-seqopt} \text{ declarator };
\]

alias-declaration:
  \[
  \text{using identifier attribute-specifier-seqopt = defining-type-id };
\]

simple-declaration:
  \[
  \text{decl-specifier-seq init-declarator-listopt };
  \text{attribute-specifier-seqopt decl-specifier-seq init-declarator-listopt };
  \text{attribute-specifier-seqopt ref-qualifieropt [ identifier-list ] initializer };
\]

static_assert-declaration:
  \[
  \text{static assert ( constant-expression )};
  \text{static assert ( constant-expression , string-literal )};
\]

empty-declaration:
  \[
  ;
\]

attribute-declaration:
  \[
  \text{attribute-specifier-seqopt };
\]

[\text{Note 1: asm-declarations are described in 9.10, and linkage-specifications are described in 9.11; function-declarations are described in 9.5 and template-declarations and deduction-guides are described in 13.7.2.3; namespace-declarations are described in 9.8.2, using-declarations are described in 9.9 and using-directives are described in 9.8.4. — end note}]

2 A simple-declaration or nodeclspec-function-declaration of the form

\[
\text{attribute-specifier-seqopt decl-specifier-seqopt init-declarator-listopt };
\]

is divided into three parts. Attributes are described in 9.12. decl-specifiers, the principal components of a decl-specifier-seq, are described in 9.2. declarators, the components of an init-declarator-list, are described in 9.3. The attribute-specifier-seq appertains to each of the entities declared by the declarators of the init-declarator-list.
A declaration occurs in a scope (6.4); the scope rules are summarized in 6.5. A declaration that declares a function or defines a class, namespace, template, or function also has one or more scopes nested within it. These nested scopes, in turn, can have declarations nested within them. Unless otherwise stated, utterances in Clause 9 about components in, of, or contained by a declaration or subcomponent thereof refer only to those components of the declaration that are not nested within scopes nested within the declaration.

In a simple-declaration, the optional init-declarator-list can be omitted only when declaring a class (Clause 11) or enumeration (9.7.1), that is, when the decl-specifier-seq contains either a class-specifier, an elaborated-type-specifier with a class-key (11.3), or an enum-specifier. In these cases and whenever a class-specifier or enum-specifier is present in the decl-specifier-seq, the identifiers in these specifiers are among the names being declared by the declaration (as class-names, enum-names, or enumerators, depending on the syntax). In such cases, the decl-specifier-seq shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration.

An empty-declaration has no effect.

A simple-declaration with an identifier-list is called a structured binding declaration (9.6). If the decl-specifier-seq contains any decl-specifier other than static, thread_local, auto (9.2.9.6), or cv-qualifiers, the program is ill-formed. The initializer shall be of the form "= assignment-expression", of the form " { assignment-expression }", or of the form " ( assignment-expression )", where the assignment-expression is of array or non-union class type.

Each init-declarator in the init-declarator-list contains exactly one declarator-id, which is the name declared by that init-declarator and hence one of the names declared by the declaration. The defining-type-specifiers (9.2.9) in the decl-specifier-seq and the recursive declarator structure of the init-declarator describe a type (9.3.4), which is then associated with the name being declared by the init-declarator.

If the decl-specifier-seq contains the typedef specifier, the declaration is called a typedef declaration and the name of each init-declarator is declared to be a typedef-name, synonymous with its associated type (9.2.4). If the decl-specifier-seq contains no typedef specifier, the declaration is called a function declaration if the type associated with the name is a function type (9.3.4.6) and an object declaration otherwise.

Syntactic components beyond those found in the general form of declaration are added to a function declaration to make a function-definition. An object declaration, however, is also a definition unless it contains the extern specifier and has no initializer (6.2). An object definition causes storage of appropriate size and alignment to be reserved and any appropriate initialization (9.4) to be done.

A nodeclspec-function-declaration shall declare a constructor, destructor, or conversion function.

[Note 3: A nodeclspec-function-declaration can only be used in a template-declaration (13.1), explicit-instantiation (13.9.3), or explicit-specialization (13.9.4). — end note]


9.2 Specifiers

9.2.1 General

The specifiers that can be used in a declaration are

\[
\text{decl-specifier:} \\
\text{storage-class-specifier} \\
\text{defining-type-specifier} \\
\text{function-specifier} \\
\text{friend} \\
\text{typedef} \\
\text{constexpr} \\
\text{consteval} \\
\text{constinit} \\
\text{inline}
\]

\[
\text{decl-specifier-seq:} \\
\text{decl-specifier attribute-specifier-seq opt} \\
\text{decl-specifier decl-specifier-seq}
\]

The optional attribute-specifier-seq in a decl-specifier-seq appertains to the type determined by the preceding decl-specifiers (9.3.4). The attribute-specifier-seq affects the type only for the declaration it appears in, not other declarations involving the same type.

At most one storage-class-specifier shall appear in a given decl-specifier-seq, except that long may appear twice. Each decl-specifier shall appear at most once in a complete decl-specifier-seq, except that constexpr, consteval, and constinit keywords shall appear in a decl-specifier-seq if and only if there is no previous defining-type-specifier other than a cv-qualifier in the decl-specifier-seq. The sequence shall be self-consistent as described below.

[Example 1:
  
  typedef char* Pc;                     // error: name missing
  static Pc;

Here, the declaration static Pc is ill-formed because no name was specified for the static variable of type Pc. To get a variable called Pc, a type-specifier (other than const or volatile) has to be present to indicate that the typedef-name Pc is the name being (re)declared, rather than being part of the decl-specifier sequence. For another example,

  void f(const Pc);                   // void f(char* const) (not const char*)
  void g(const int Pc);              // void g(const int)

—end example]

[Note 1: Since signed, unsigned, long, and short by default imply int, a type-name appearing after one of those specifiers is treated as the name being (re)declared.

[Example 2:
  
  void h(unsigned Pc);                // void h(unsigned int)
  void k(unsigned int Pc);           // void k(unsigned int)

—end example]

—end note]

9.2.2 Storage class specifiers

The storage class specifiers are

\[
\text{storage-class-specifier:} \\
\text{static} \\
\text{thread_local} \\
\text{extern} \\
\text{mutable}
\]

At most one storage-class-specifier shall appear in a given decl-specifier-seq, except that thread_local may appear with static or extern. If thread_local appears in any declaration of a variable it shall be present in all declarations of that entity. If a storage-class-specifier appears in a decl-specifier-seq, there can be no typedef specifier in the same decl-specifier-seq and the init-declarator-list or member-declarator-list of the declaration shall not be empty (except for an anonymous union declared in a named namespace or in the global namespace, which shall be declared static (11.5.2)). The storage-class-specifier applies to the name declared by each init-declarator in the list and not to any names declared by other specifiers.
[Note 1: See 13.9.4 and 13.9.3 for restrictions in explicit specializations and explicit instantiations, respectively. — end note]

[Note 2: A variable declared without a storage-class-specifier at block scope or declared as a function parameter has automatic storage duration by default (6.7.5.4). — end note]

3 The thread_local specifier indicates that the named entity has thread storage duration (6.7.5.3). It shall be applied only to the declaration of a variable of namespace or block scope, to a structured binding declaration (9.6), or to the declaration of a static data member. When thread_local is applied to a variable of block scope the storage-class-specifier static is implied if no other storage-class-specifier appears in the decl-specifier-seq.

4 The static specifier shall be applied only to the declaration of a variable or function, to a structured binding declaration (9.6), or to the declaration of an anonymous union (11.5.2). There can be no static function declarations within a block, nor any static function parameters. A static specifier used in the declaration of a variable declares the variable to have static storage duration (6.7.5.2), unless accompanied by the thread_local specifier, which declares the variable to have thread storage duration (6.7.5.3). A static specifier can be used in declarations of class members; 11.4.9 describes its effect. For the linkage of a name declared with a static specifier, see 6.6.

5 The extern specifier shall be applied only to the declaration of a variable or function. The extern specifier shall not be used in the declaration of a class member or function parameter. For the linkage of a name declared with an extern specifier, see 6.6.

[Note 3: The extern keyword can also be used in explicit-instantiations and linkage-specifications, but it is not a storage-class-specifier in such contexts. — end note]

6 The linkages implied by successive declarations for a given entity shall agree. That is, within a given scope, each declaration declaring the same variable name or the same overloading of a function name shall imply the same linkage.

[Example 1:

```cpp
static char* f(); // f() has internal linkage
char* f(); // f() still has internal linkage
{ /* ... */ }
char* g(); // g() has external linkage
static char* g() // error: inconsistent linkage
{ /* ... */ }
void h();
inline void h(); // external linkage
inline void l(); // external linkage
void l();
inline void m(); // external linkage
extern void m();
static void n();
inline void n(); // internal linkage
static int a; // a has internal linkage
int a; // error: two definitions
static int b; // b has internal linkage
extern int b; // b still has internal linkage
int c; // c has external linkage
static int c; // error: inconsistent linkage
extern int d; // d has external linkage
static int d; // error: inconsistent linkage
``` — end example]
The name of a declared but undefined class can be used in an `extern` declaration. Such a declaration can only be used in ways that do not require a complete class type.

[Example 2:
```c
struct S;
extern S a;
extern S f();
extern void g(S);

void h() {
  g(a); // error: S is incomplete
  f(); // error: S is incomplete
}
```
—end example]

The `mutable` specifier shall appear only in the declaration of a non-static data member (11.4) whose type is neither const-qualified nor a reference type.

[Example 3:
```c
class X {
  mutable const int* p; // OK
  mutable int* const q; // error
};
```
—end example]

[Note 4: The `mutable` specifier on a class data member nullifies a `const` specifier applied to the containing class object and permits modification of the mutable class member even though the rest of the object is `const` (6.8.4, 9.2.9.2). —end note]

### 9.2.3 Function specifiers

A `function-specifier` can be used only in a function declaration.

```c
function-specifier:
  virtual
  explicit-specifier
  explicit ( constant-expression )
  explicit
```

The `virtual` specifier shall be used only in the initial declaration of a non-static member function; see 11.7.3.

An `explicit-specifier` shall be used only in the declaration of a constructor or conversion function within its class definition; see 11.4.8.2 and 11.4.8.3.

In an `explicit-specifier`, the `constant-expression`, if supplied, shall be a contextually converted constant expression of type `bool` (7.7). The `explicit-specifier explicit` without a `constant-expression` is equivalent to the `explicit-specifier explicit(true)`. If the constant expression evaluates to `true`, the function is explicit. Otherwise, the function is not explicit. A `(` token that follows `explicit` is parsed as part of the `explicit-specifier`.

### 9.2.4 The `typedef` specifier

Declarations containing the `decl-specifier `typedef` declare identifiers that can be used later for naming fundamental (6.8.2) or compound (6.8.3) types. The `typedef` specifier shall not be combined in a `decl-specifier-seq` with any other kind of specifier except a `defining-type-specifier`, and it shall not be used in the `decl-specifier-seq` of a `parameter-declaration` (9.3.4.6) nor in the `decl-specifier-seq` of a `function-definition` (9.5). If a `typedef` specifier appears in a declaration without a `declarator`, the program is ill-formed.

```c
typedef-name:
  identifier
  simple-template-id
```

A name declared with the `typedef` specifier becomes a `typedef-name`. A `typedef-name` names the type associated with the `identifier` (9.3) or `simple-template-id` (13.1); a `typedef-name` is thus a synonym for another type. A `typedef-name` does not introduce a new type the way a class declaration (11.3) or `enum` declaration (9.7.1) does.

[Example 1: After...
typedef int MILES, *KLICKSP;
The constructions

MILES distance;
extern KLICKSP metricp;

are all correct declarations; the type of distance is int and that of metricp is “pointer to int”. — end example]

A typedef-name can also be introduced by an alias-declaration. The identifier following the using keyword becomes a typedef-name and the optional attribute-specifier-seq following the identifier appertains to that typedef-name. Such a typedef-name has the same semantics as if it were introduced by the typedef specifier. In particular, it does not define a new type.

Example 2:

using handler_t = void (*)(int);
extern handler_t ignore;
extern void (*ignore)(int);          // redeclare ignore
using cell = pair<void*, cell*>;      // error
— end example]

The defining-type-specifier-seq of the defining-type-id shall not define a class or enumeration if the alias-declaration is the declaration of a template-declaration.

In a given non-class scope, a typedef specifier can be used to redeclare the name of any type declared in that scope to refer to the type to which it already refers.

Example 3:

typedef struct s { /* ... */ } s;
typedef int I;
typedef I I;
— end example]

In a given class scope, a typedef specifier can be used to redeclare any class-name declared in that scope that is not also a typedef-name to refer to the type to which it already refers.

Example 4:

struct S {
  typedef struct A {} A;        // OK
  typedef struct B B;           // OK
  typedef A A;                  // error
};
— end example]

If a typedef specifier is used to redeclare in a given scope an entity that can be referenced using an elaborated-type-specifier, the entity can continue to be referenced by an elaborated-type-specifier or as an enumeration or class name in an enumeration or class definition respectively.

Example 5:

struct S;
typedef struct S S;
int main() {
  struct S* p;                 // OK
}
struct S {}                     // OK
— end example]

In a given scope, a typedef specifier shall not be used to redeclare the name of any type declared in that scope to refer to a different type.

Example 6:

class complex { /* ... */};
typedef int complex;            // error: redefinition
— end example]
Similarly, in a given scope, a class or enumeration shall not be declared with the same name as a typedef-name that is declared in that scope and refers to a type other than the class or enumeration itself.

[Example 7:
    typedef int complex;
    class complex { /* ... */ };  // error: redefinition
    — end example]

A simple-template-id is only a typedef-name if its template-name names an alias template or a template template-parameter.

[Note 1: A simple-template-id that names a class template specialization is a class-name (11.3). If a typedef-name is used to identify the subject of an elaborated-type-specifier (9.2.9.4), a class definition (Clause 11), a constructor declaration (11.4.5), or a destructor declaration (11.4.7), the program is ill-formed. — end note]

[Example 8:
    struct S {
        S();
        ’-S();
    };
    typedef struct S T;
    S a = T();  // OK
    struct T * p;  // error
    — end example]

If the typedef declaration defines an unnamed class or enumeration, the first typedef-name declared by the declaration to be that type is used to denote the type for linkage purposes only (6.6).

[Note 2: A typedef declaration involving a lambda-expression does not itself define the associated closure type, and so the closure type is not given a name for linkage purposes. — end note]

[Example 9:
    typedef struct { } *ps, S;  // S is the class name for linkage purposes
    typedef decltype(1{}) C;  // the closure type has no name for linkage purposes
    — end example]

An unnamed class with a typedef name for linkage purposes shall not

(10.1) — declare any members other than non-static data members, member enumerations, or member classes,

(10.2) — have any base classes or default member initializers, or

(10.3) — contain a lambda-expression,

and all member classes shall also satisfy these requirements (recursively).

[Example 10:
    typedef struct {
        int f() {}
    } X;  // error: struct with typedef name for linkage has member functions
    — end example]

9.2.5 The friend specifier [dcl.friend]

The friend specifier is used to specify access to class members; see 11.9.4.

9.2.6 The constexpr and consteval specifiers [dcl.constexpr]

The constexpr specifier shall be applied only to the definition of a variable or variable template or the declaration of a function or function template. The consteval specifier shall be applied only to the declaration of a function or function template. A function or static data member declared with the constexpr or consteval specifier is implicitly an inline function or variable (9.2.8). If any declaration of a function or function template has a constexpr or consteval specifier, then all its declarations shall contain the same specifier.

[Note 1: An explicit specialization can differ from the template declaration with respect to the constexpr or consteval specifier. — end note]
[Note 2: Function parameters cannot be declared constexpr. — end note]

[Example 1:

```cpp
constexpr void square(int &x); // OK: declaration
cconstexpr int bufSz = 1024; // OK: definition
cconstexpr struct pixel {
    int x;
    int y;
    constexpr pixel(int); // OK: declaration
};
cconstexpr pixel::pixel(int a) : x(a), y(x) // OK: definition
{ square(x); }
cconstexpr pixel small(2); // error: square not defined, so small(2) // not constant (7.7) so constexpr not satisfied

cconstexpr void square(int &x) { // OK: definition
    x *= x;
}
cconstexpr pixel large(4); // OK: square defined
int next(constexpr int x) { // error: not for parameters
    return x + 1;
}
extern constexpr int memsz; // error: not a definition
— end example]

2 A constexpr or consteval specifier used in the declaration of a function declares that function to be a constexpr function. A function or constructor declared with the consteval specifier is called an immediate function. A destructor, an allocation function, or a deallocation function shall not be declared with the consteval specifier.

3 The definition of a constexpr function shall satisfy the following requirements:

(3.1) — its return type (if any) shall be a literal type;
(3.2) — each of its parameter types shall be a literal type;
(3.3) — it shall not be a coroutine (9.5.4);
(3.4) — if the function is a constructor or destructor, its class shall not have any virtual base classes;
(3.5) — its function-body shall not enclose (8.1)
(3.5.1) — a goto statement,
(3.5.2) — an identifier label (8.2),
(3.5.3) — a definition of a variable of non-literal type or of static or thread storage duration.

[Note 3: A function-body that is = delete or = default encloses none of the above. — end note]

[Example 2:

```cpp
constexpr int square(int x) {
    return x * x; }
constexpr long long_max() {
    return 2147483647; }
constexpr int abs(int x) {
    if (x < 0)
        x = -x;
    return x;
}
constexpr int first(int n) {
    static int value = n;
    return value;
}
constexpr int uninit() {
    struct { int a; } s;
    return s.a; // error: uninitialized read of s.a
}
```
constexpr int prev(int x)
{ return --x; }  // OK
constexpr int g(int x, int n) {  // OK
    int r = 1;
    while (--n > 0) r *= x;
    return r;
}
—end example]

4 The definition of a constexpr constructor whose function-body is not = delete shall additionally satisfy the following requirements:

(4.1) — for a non-delegating constructor, every constructor selected to initialize non-static data members and base class subobjects shall be a constexpr constructor;

(4.2) — for a delegating constructor, the target constructor shall be a constexpr constructor.

[Example 3:
struct Length {
    constexpr explicit Length(int i = 0) : val(i) { }
private:
    int val;
};  
—end example]

5 The definition of a constexpr destructor whose function-body is not = delete shall additionally satisfy the following requirement:

(5.1) — for every subobject of class type or (possibly multi-dimensional) array thereof, that class type shall have a constexpr destructor.

6 For a constexpr function or constexpr constructor that is neither defaulted nor a template, if no argument values exist such that an invocation of the function or constructor could be an evaluated subexpression of a core constant expression (7.7), or, for a constructor, an evaluated subexpression of the initialization full-expression of some constant-initialized object (6.9.3.2), the program is ill-formed, no diagnostic required.

[Example 4:
constexpr int f(bool b)
{ return b ? throw 0 : 0; }  // OK
constexpr int f() { return f(true); }  // ill-formed, no diagnostic required

struct B {
    constexpr B(int x) : i(0) { }  // x is unused
    int i;
};

int global;

struct D : B {
    constexpr D() : B(global) { }  // ill-formed, no diagnostic required
    // value-to-­value conversion on non-­constant global
};  
—end example]

7 If the instantiated template specialization of a constexpr function template or member function of a class template would fail to satisfy the requirements for a constexpr function, that specialization is still a constexpr function, even though a call to such a function cannot appear in a constant expression. If no specialization of the template would satisfy the requirements for a constexpr function when considered as a non-template function, the template is ill-formed, no diagnostic required.

8 An invocation of a constexpr function in a given context produces the same result as an invocation of an equivalent non-constexpr function in the same context in all respects except that

(8.1) — an invocation of a constexpr function can appear in a constant expression (7.7) and

(8.2) — copy elision is not performed in a constant expression (11.10.6).
The `constexpr` and `consteval` specifiers have no effect on the type of a constexpr function.

**Example 5:**
```cpp
castexpr int bar(int x, int y) // OK
{ return x + y + x*y; }
//...
int bar(int x, int y) // error: redefinition of bar
{ return x * 2 + 3 * y; }
```

A `constexpr` specifier used in an object declaration declares the object as const. Such an object shall have literal type and shall be initialized. In any `constexpr` variable declaration, the full-expression of the initialization shall be a constant expression (7.7). A `constexpr` variable shall have constant destruction.

**Example 6:**
```cpp
struct pixel {
    int x, y;
};
castexpr pixel ur = { 1294, 1024 }; // OK
constexpr pixel origin; // error: initializer missing
```

### 9.2.7 The `constinit` specifier

The `constinit` specifier shall be applied only to a declaration of a variable with static or thread storage duration. If the specifier is applied to any declaration of a variable, it shall be applied to the initializing declaration. No diagnostic is required if no `constinit` declaration is reachable at the point of the initializing declaration.

If a variable declared with the `constinit` specifier has dynamic initialization (6.9.3.3), the program is ill-formed.

**Example 1:**
```cpp
const char * g() { return "dynamic initialization"; }
castexpr const char * f(bool p) { return p ? "constant initializer" : g(); }
constinit const char * c = f(true); // OK
constinit const char * d = f(false); // error
```

### 9.2.8 The `inline` specifier

The `inline` specifier shall be applied only to the declaration of a variable or function.

A function declaration (9.3.4.6, 11.4.2, 11.9.4) with an `inline` specifier declares an inline function. The `inline` specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism. An implementation is not required to perform this inline substitution at the point of call; however, even if this inline substitution is omitted, the other rules for inline functions specified in this subclause shall still be respected.

A variable declaration with an `inline` specifier declares an inline variable.

The `inline` specifier shall not appear on a block scope declaration or on the declaration of a function parameter. If the `inline` specifier is used in a friend function declaration, that declaration shall be a definition or the function shall have previously been declared inline.

If a definition of a function or variable is reachable at the point of its first declaration as inline, the program is ill-formed. If a function or variable with external or module linkage is declared inline in one definition
domain, an inline declaration of it shall be reachable from the end of every definition domain in which it is declared; no diagnostic is required.

[Note 2: A call to an inline function or a use of an inline variable can be encountered before its definition becomes reachable in a translation unit. — end note]

[Note 3: An inline function or variable with external or module linkage has the same address in all translation units. A static local variable in an inline function with external or module linkage always refers to the same object. A type defined within the body of an inline function with external or module linkage is the same type in every translation unit. — end note]

If an inline function or variable that is attached to a named module is declared in a definition domain, it shall be defined in that domain.

[Note 4: A constexpr function (9.2.6) is implicitly inline. In the global module, a function defined within a class definition is implicitly inline (11.4.2, 11.9.4). — end note]

9.2.9 Type specifiers

9.2.9.1 General

The type-specifiers are

- type-specifier:
  - simple-type-specifier
  - elaborated-type-specifier
  - typename-specifier
  - cv-qualifier

- type-specifier-seq:
  - type-specifier attribute-specifier-seq_opt
  - type-specifier type-specifier-seq

- defining-type-specifier:
  - type-specifier
  - class-specifier
  - enum-specifier

- defining-type-specifier-seq:
  - defining-type-specifier attribute-specifier-seq_opt
  - defining-type-specifier defining-type-specifier-seq

The optional attribute-specifier-seq in a type-specifier-seq or a defining-type-specifier-seq appertains to the type denoted by the preceding type-specifiers or defining-type-specifiers (9.3.4). The attribute-specifier-seq affects the type only for the declaration it appears in, not other declarations involving the same type.

As a general rule, at most one defining-type-specifier is allowed in the complete decl-specifier-seq of a declaration or in a defining-type-specifier-seq, and at most one type-specifier is allowed in a type-specifier-seq. The only exceptions to this rule are the following:

1. const can be combined with any type specifier except itself.
2. volatile can be combined with any type specifier except itself.
3. signed or unsigned can be combined with char, long, short, or int.
4. short or long can be combined with int.
5. long can be combined with double.
6. long can be combined with long.

Except in a declaration of a constructor, destructor, or conversion function, at least one defining-type-specifier that is not a cv-qualifier shall appear in a complete type-specifier-seq or a complete decl-specifier-seq.

[Note 1: enum-specifiers, class-specifiers, and typename-specifiers are discussed in 9.7.1, Clause 11, and 13.8, respectively. The remaining type-specifiers are discussed in the rest of 9.2.9. — end note]

9.2.9.2 The cv-qualifiers

There are two cv-qualifiers, const and volatile. Each cv-qualifier shall appear at most once in a cv-qualifier-seq. If a cv-qualifier appears in a decl-specifier-seq, the init-declarator-list or member-declarator-list of the declaration shall not be empty.

— const can be combined with any type specifier except itself.
— volatile can be combined with any type specifier except itself.
— signed or unsigned can be combined with char, long, short, or int.
— short or long can be combined with int.
— long can be combined with double.
— long can be combined with long.

Except in a declaration of a constructor, destructor, or conversion function, at least one defining-type-specifier that is not a cv-qualifier shall appear in a complete type-specifier-seq or a complete decl-specifier-seq.

[Note 1: enum-specifiers, class-specifiers, and typename-specifiers are discussed in 9.7.1, Clause 11, and 13.8, respectively. The remaining type-specifiers are discussed in the rest of 9.2.9. — end note]
[Note 1: 6.8.4 and 9.3.4.6 describe how cv-qualifiers affect object and function types. — end note]

Redundant cv-qualifications are ignored.

[Note 2: For example, these could be introduced by typedefs. — end note]

2 [Note 2: Declaring a variable const can affect its linkage (9.2.2) and its usability in constant expressions (7.7). As described in 9.4, the definition of an object or subobject of const-qualified type must specify an initializer or be subject to default-initialization. — end note]

3 A pointer or reference to a cv-qualified type need not actually point or refer to a cv-qualified object, but it is treated as if it does; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path.

[Note 1: Cv-qualifiers are supported by the type system so that they cannot be subverted without casting (7.6.1.11). — end note]

4 Any attempt to modify (7.6.19, 7.6.1.6, 7.6.2.3) a const object (6.8.4) during its lifetime (6.7.3) results in undefined behavior.

[Example 1:]
```cpp
class X {  
mutable int i;
    int j;
};

class Y {  
    X x;
    Y();
};

const Y y;
y.x.i++;  // well-formed: mutable member can be modified
y.x.j++;  // error: const-qualified member modified
Y* p = const_cast<Y*>(&y);  // cast away const-ness of y
p->x.i = 99;  // well-formed: mutable member can be modified
p->x.j = 99;  // undefined behavior: modifies a const subobject
```

5 The semantics of an access through a volatile glvalue are implementation-defined. If an attempt is made to access an object defined with a volatile-qualified type through the use of a non-volatile glvalue, the behavior is undefined.

6 [Note 5: volatile is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. Furthermore, for some implementations, volatile might indicate that special hardware instructions are required to access the object. See 6.9.1 for detailed semantics. In general, the semantics of volatile are intended to be the same in C++ as they are in C. — end note]

9.2.9.3 Simple type specifiers [dcl.type.simple]

1 The simple type specifiers are
simple-type-specifier:
   nested-name-specifier_opt type-name
   nested-name-specifier template simple-template-id
dectype-specifier
placeholder-type-specifier
   nested-name-specifier_opt template-name
   char
cchar_t
cchar32_t
wchar_t
   bool
   short
   int
   long
   signed
   unsigned
   float
   double
   void
type-name:
   class-name
   enum-name
typedef-name

A placeholder-type-specifier is a placeholder for a type to be deduced (9.2.9.6). A type-specifier of the form
typename_opt nested-name-specifier_opt template-name is a placeholder for a deduced class type (9.2.9.7). The
nested-name-specifier, if any, shall be non-dependent and the template-name shall name a deducible template.
A deducible template is either a class template or is an alias template whose defining-type-id is of the form
typename_opt nested-name-specifier_opt template_opt simple-template-id
where the nested-name-specifier (if any) is non-dependent and the template-name of the simple-template-id
names a deducible template.

[Note 1: An injected-class-name is never interpreted as a template-name in contexts where class template argument
deduction would be performed (13.8.2). — end note]

The other simple-type-specifiers specify either a previously-declared type, a type determined from an expression,
or one of the fundamental types (6.8.2). Table 14 summarizes the valid combinations of simple-type-specifiers
and the types they specify.

When multiple simple-type-specifiers are allowed, they can be freely intermixed with other decl-specifiers in
any order.

[Note 2: It is implementation-defined whether objects of char type are represented as signed or unsigned quantities.
The signed specifier forces char objects to be signed; it is redundant in other contexts. — end note]

9.2.9.4 Elaborated type specifiers  [dcl.type.elab]

elaborated-type-specifier:
   class-key attribute-specifier-seq_opt nested-name-specifier_opt identifier
class-key simple-template-id
class-key nested-name-specifier template_opt simple-template-id
   elaborated-enum-specifier

elaborated-enum-specifier:
   enum nested-name-specifier_opt identifier

An attribute-specifier-seq shall not appear in an elaborated-type-specifier unless the latter is the sole constituent
of a declaration. If an elaborated-type-specifier is the sole constituent of a declaration, the declaration is
ill-formed unless it is an explicit specialization (13.9.4), an explicit instantiation (13.9.3) or it has one of the
following forms:
class-key attribute-specifier-seq_opt identifier ;
friend class-key ::opt identifier ;
friend class-key ::opt simple-template-id ;
friend class-key nested-name-specifier identifier ;
friend class-key nested-name-specifier template_opt simple-template-id ;
Table 14: *simple-type-specifiers* and the types they specify  

<table>
<thead>
<tr>
<th>Specifier(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>type-name</td>
<td>the type named</td>
</tr>
<tr>
<td>simple-template-id</td>
<td>the type as defined in 13.3</td>
</tr>
<tr>
<td>decltype-specifier</td>
<td>the type as defined in 9.2.9.5</td>
</tr>
<tr>
<td>placeholder-type-specifier</td>
<td>the type as defined in 9.2.9.6</td>
</tr>
<tr>
<td>template-name</td>
<td>the type as defined in 9.2.9.7</td>
</tr>
<tr>
<td>char</td>
<td>“char”</td>
</tr>
<tr>
<td>unsigned char</td>
<td>“unsigned char”</td>
</tr>
<tr>
<td>signed char</td>
<td>“signed char”</td>
</tr>
<tr>
<td>char8_t</td>
<td>“char8_t”</td>
</tr>
<tr>
<td>char16_t</td>
<td>“char16_t”</td>
</tr>
<tr>
<td>char32_t</td>
<td>“char32_t”</td>
</tr>
<tr>
<td>bool</td>
<td>“bool”</td>
</tr>
<tr>
<td>unsigned int</td>
<td>“unsigned int”</td>
</tr>
<tr>
<td>signed int</td>
<td>“int”</td>
</tr>
<tr>
<td>int</td>
<td>“int”</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>“unsigned short int”</td>
</tr>
<tr>
<td>unsigned short</td>
<td>“unsigned short int”</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>“unsigned long int”</td>
</tr>
<tr>
<td>unsigned long</td>
<td>“unsigned long int”</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>“unsigned long long int”</td>
</tr>
<tr>
<td>signed long int</td>
<td>“long int”</td>
</tr>
<tr>
<td>signed long</td>
<td>“long int”</td>
</tr>
<tr>
<td>signed long long int</td>
<td>“long long int”</td>
</tr>
<tr>
<td>signed long long</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long long</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long</td>
<td>“long int”</td>
</tr>
<tr>
<td>signed short int</td>
<td>“short int”</td>
</tr>
<tr>
<td>signed short</td>
<td>“short int”</td>
</tr>
<tr>
<td>short int</td>
<td>“short int”</td>
</tr>
<tr>
<td>short</td>
<td>“short int”</td>
</tr>
<tr>
<td>wchar_t</td>
<td>“wchar_t”</td>
</tr>
<tr>
<td>float</td>
<td>“float”</td>
</tr>
<tr>
<td>double</td>
<td>“double”</td>
</tr>
<tr>
<td>long double</td>
<td>“long double”</td>
</tr>
<tr>
<td>void</td>
<td>“void”</td>
</tr>
</tbody>
</table>

In the first case, the *attribute-specifier-seq*, if any, appertains to the class being declared; the attributes in the *attribute-specifier-seq* are thereafter considered attributes of the class whenever it is named.

2 [Note 1: 6.5.5 describes how name lookup proceeds for the *identifier* in an *elaborated-type-specifier*. — end note]

If the *identifier* or *simple-template-id* resolves to a *class-name* or *enum-name*, the *elaborated-type-specifier* introduces it into the declaration the same way a *simple-type-specifier* introduces its *type-name* (9.2.9.3). If the *identifier* or *simple-template-id* resolves to a *typedef-name* (9.2.4, 13.3), the *elaborated-type-specifier* is ill-formed.

[Note 2: This implies that, within a class template with a template *type-parameter* T, the declaration

```cpp
friend class T;
```

is ill-formed. However, the similar declaration `friend T;` is allowed (11.9.4). — end note]
The class-key or enum keyword present in the elaborated-type-specifier shall agree in kind with the declaration to which the name in the elaborated-type-specifier refers. This rule also applies to the form of elaborated-type-specifier that declares a class-name or friend class since it can be construed as referring to the definition of the class. Thus, in any elaborated-type-specifier, the enum keyword shall be used to refer to an enumeration (9.7.1), the union class-key shall be used to refer to a union (11.5), and either the class or struct class-key shall be used to refer to a non-union class (11.1).

Example 1:
```c
enum class E { a, b };  
enum E x = E::a;  // OK
struct S { } s;  
class S* p = &s;  // OK
```

9.2.9.5 Decltype specifiers

dcl.type.decltype
dcl.type.decltype:
dcl.typeSpecifier ( dcl.type.decltype )

For an expression E, the type denoted by decltype(E) is defined as follows:

1. if E is an unparenthesized id-expression naming a structured binding (9.6), decltype(E) is the referenced type as given in the specification of the structured binding declaration;
2. otherwise, if E is an unparenthesized id-expression naming a non-type template-parameter (13.2), decltype(E) is the type of the template-parameter after performing any necessary type deduction (9.2.9.6, 9.2.9.7);
3. otherwise, if E is an unparenthesized id-expression or an unparenthesized class member access (7.6.1.5), decltype(E) is the type of the entity named by E. If there is no such entity, or if E names a set of overloaded functions, the program is ill-formed;
4. otherwise, if E is an xvalue, decltype(E) is T&&, where T is the type of E;
5. otherwise, if E is an lvalue, decltype(E) is T&, where T is the type of E;
6. otherwise, decltype(E) is the type of E.

The operand of the decltype specifier is an unevaluated operand (7.2).

Example 1:
```c
const int&& foo();
int i;
struct A { double x; };  
const A* a = new A();
dectype(foo()) x1 = 17;  // type is const int&
dectype(i) x2;  // type is int
dectype(a->x) x3;  // type is double
dectype((a->x)) x4 = x3;  // type is const double&
```

Note 1: The rules for determining types involving decltype(auto) are specified in 9.2.9.6. — end note

If the operand of a decltype-specifier is a prvalue and is not a (possibly parenthesized) immediate invocation (7.7), the temporary materialization conversion is not applied (7.3.5) and no result object is provided for the prvalue. The type of the prvalue may be incomplete or an abstract class type.

Note 2: As a result, storage is not allocated for the prvalue and it is not destroyed. Thus, a class type is not instantiated as a result of being the type of a function call in this context. In this context, the common purpose of writing the expression is merely to refer to its type. In that sense, a decltype-specifier is analogous to a use of a typedef-name, so the usual reasons for requiring a complete type do not apply. In particular, it is not necessary to allocate storage for a temporary object or to enforce the semantic constraints associated with invoking the type’s destructor. — end note

Note 3: Unlike the preceding rule, parentheses have no special meaning in this context. — end note

Example 2:
```c
template<class T> struct A { ~A() = delete; };  
template<class T> auto h()  
    -> A<T>;
```
template<class T> auto i(T) // identity
    -> T;
template<class T> auto f(T) // #1
    -> decltype(i(h<T>())); // forces completion of A<T> and implicitly uses A<T>::~A()
    // for the temporary introduced by the use of h().
    // (A temporary is not introduced as a result of the use of i().)
template<class T> auto f(T) // #2
    -> void;
auto g() -> void {
    f(42); // OK: calls #2. (#1 is not a viable candidate: type deduction
    // fails (13.10.3) because A<int>::~A() is implicitly used in its
    // decltype-specifier)
}

—end example]

9.2.9.6 Placeholder type specifiers [dcl.spec.auto]
9.2.9.6.1 General [dcl.spec.auto.general]

placeholder-type-specifier:
    type-constraint\opt auto
type-constraint\opt decltype ( auto )

1 A placeholder-type-specifier designates a placeholder type that will be replaced later by deduction from an
    initializer.

2 A placeholder-type-specifier of the form type-constraint\opt auto can be used as a decl-specifier of the
    decl-specifier-seq of a parameter-declaration of a function declaration or lambda-expression and, if it is not the auto
    type-specifier introducing a trailing-return-type (see below), is a generic parameter type placeholder of the
    function declaration or lambda-expression.

[Note 1: Having a generic parameter type placeholder signifies that the function is an abbreviated function template
(9.3.4.6) or the lambda is a generic lambda (7.5.5). — end note]

3 The placeholder type can appear with a function declarator in the decl-specifier-seq, type-specifier-seq,
    conversion-function-id, or trailing-return-type, in any context where such a declarator is valid. If the function
    declarator includes a trailing-return-type (9.3.4.6), that trailing-return-type specifies the declared return type of
    the function. Otherwise, the function declarator shall declare a function. If the declared return type of the
    function contains a placeholder type, the return type of the function is deduced from non-discarded return
    statements, if any, in the body of the function (8.5.2).

4 The type of a variable declared using a placeholder type is deduced from its initializer. This use is allowed in
   an initializing declaration (9.4) of a variable. The placeholder type shall appear as one of the decl-specifiers in
   the decl-specifier-seq and the decl-specifier-seq shall be followed by one or more decls, each of which shall
   be followed by a non-empty initializer. If the initializer is a parenthesized expression-list, the expression-list
   shall be a single assignment-expression.

[Example 1:

auto x = 5; // OK: x has type int
const auto *v = &x, u = 6; // OK: v has type const int*, u has type const int
static auto y = 0.0; // OK: y has type double
auto int r; // error: auto is not a storage-class-specifier
auto f() -> int; // OK: f returns int
auto g() { return 0.0; } // OK: g returns double
auto h(); // OK: h's return type will be deduced when it is defined

— end example]

The auto type-specifier can also be used to introduce a structured binding declaration (9.6).
A placeholder type can also be used in the type-specifier-seq in the new-type-id or type-id of a new-expression (7.6.2.8) and as a decl-specifier of the parameter-declaration’s decl-specifier-seq in a template-parameter (13.2).

A program that uses a placeholder type in a context not explicitly allowed in 9.2.9.6 is ill-formed.

If the init-declarator-list contains more than one init-declarator, they shall all form declarations of variables. The type of each declared variable is determined by placeholder type deduction (9.2.9.6.2), and if the type that replaces the placeholder type is not the same in each deduction, the program is ill-formed.

[Example 2:
  auto x = 5, *y = &x;  // OK: auto is int
  auto a = 5, b = { 1, 2 };  // error: different types for auto
— end example]

If a function with a declared return type that contains a placeholder type has multiple non-discarded return statements, the return type is deduced for each such return statement. If the type deduced is not the same in each deduction, the program is ill-formed.

If a function with a declared return type that uses a placeholder type has no non-discarded return statements, the return type is deduced as though from a return statement with no operand at the closing brace of the function body.

[Example 3:
  auto f() { }  // OK, return type is void
  auto* g() { }  // error: cannot deduce auto* from void()
— end example]

An exported function with a declared return type that uses a placeholder type shall be defined in the translation unit containing its exported declaration, outside the private-module-fragment (if any).

[Note 2: The deduced return type cannot have a name with internal linkage (6.6). — end note]

If the name of an entity with an undeduced placeholder type appears in an expression, the program is ill-formed. Once a non-discarded return statement has been seen in a function, however, the return type deduced from that statement can be used in the rest of the function, including in other return statements.

[Example 4:
  auto n = n;  // error: n’s initializer refers to n
  auto f();
  void g() { &f; }  // error: f’s return type is unknown
  auto sum(int i) {
    if (i == 1)
      return i;
    else
      return sum(i-1)+i;  // OK, sum’s return type has been deduced
  }
— end example]

Return type deduction for a templated entity that is a function or function template with a placeholder in its declared type occurs when the definition is instantiated even if the function body contains a return statement with a non-type-dependent operand.

[Note 3: Therefore, any use of a specialization of the function template will cause an implicit instantiation. Any errors that arise from this instantiation are not in the immediate context of the function type and can result in the program being ill-formed (13.10.3). — end note]

[Example 5:
  template <class T> auto f(T t) { return t; }  // return type deduced at instantiation time
  typedef decltype(f(1)) fint_t;  // instantiates f<int> to deduce return type
  template<class T> auto f(T* t) { return *t; }  // instantiates both fs to determine return types,
  void g() { int (*p)(int*) = &f; }  // chooses second
— end example]

Redeclarations or specializations of a function or function template with a declared return type that uses a placeholder type shall also use that placeholder, not a deduced type. Similarly, redeclarations or specializations
of a function or function template with a declared return type that does not use a placeholder type shall not use a placeholder.

[Example 6:

```cpp
auto f();
auto f() { return 42; } // return type is int
auto f(); // OK
int f(); // error: cannot be overloaded with auto f()
dcltype(auto) f(); // error: auto and dcltype(auto) don't match
```

```cpp
template <typename T> auto g(T t) { return t; } // #1
template auto g(int); // OK, return type is int
template char g(char); // error: no matching template
template<auto> g(double); // OK, forward declaration with unknown return type

```

```cpp
void h() { return g(42); } // error: ambiguous
```

```cpp
template <typename T> struct A {
friend T frf(T);
};
auto frf(int i) { return i; } // not a friend of A<int>
extern int v;
auto v = 17; // OK, redeclares v
struct S {
    static int i;
};
auto S::i = 23; // OK
```

— end example]

A function declared with a return type that uses a placeholder type shall not be virtual (11.7.3).
A function declared with a return type that uses a placeholder type shall not be a coroutine (9.5.4).
An explicit instantiation declaration (13.9.3) does not cause the instantiation of an entity declared using a placeholder type, but it also does not prevent that entity from being instantiated as needed to determine its type.

[Example 7:

```cpp
template <typename T> auto f(T t) { return t; }
```

```cpp
extern template auto f(int); // does not instantiate f<int>
int (*p)(int) = f; // instantiates f<int> to determine its return type, but an explicit instantiation definition is still required somewhere in the program
```

— end example]

### 9.2.9.6.2 Placeholder type deduction

**Placeholder type deduction** is the process by which a type containing a placeholder type is replaced by a deduced type.

1 A type `T` containing a placeholder type, and a corresponding initializer `E`, are determined as follows:

1.1 for a non-discarded `return` statement that occurs in a function declared with a return type that contains a placeholder type, `T` is the declared return type and `E` is the operand of the `return` statement. If the `return` statement has no operand, then `E` is `void();`

1.2 for a variable declared with a type that contains a placeholder type, `T` is the declared type of the variable and `E` is the initializer. If the initialization is direct-list-initialization, the initializer shall be a `braced-init-list` containing only a single `assignment-expression` and `E` is the `assignment-expression`;

1.3 for a non-type template parameter declared with a type that contains a placeholder type, `T` is the declared type of the non-type template parameter and `E` is the corresponding template argument.

§ 9.2.9.6.2
In the case of a return statement with no operand or with an operand of type void, \( T \) shall be either type-constraint_opt decltype(auto) or cv type-constraint_opt auto.

3 If the deduction is for a return statement and \( E \) is a braced-init-list (9.4.5), the program is ill-formed.

4 If the placeholder-type-specifier is of the form type-constraint_opt auto, the deduced type \( T' \) replacing \( T \) is determined using the rules for template argument deduction. Obtain \( P \) from \( T \) by replacing the occurrences of type-constraint_opt auto either with a new invented type template parameter \( U \) or, if the initialization is copy-list-initialization, with std::initializer_list<U>. Deduce a value for \( U \) using the rules of template argument deduction from a function call (13.10.3.2), where \( P \) is a function template parameter type and the corresponding argument is \( E \). If the deduction fails, the declaration is ill-formed. Otherwise, \( T' \) is obtained by substituting the deduced \( U \) into \( P \).

[Example 1:
```
auto x1 = { 1, 2 }; // decltype(x1) is std::initializer_list<int>
auto x2 = { 1, 2.0 }; // error: cannot deduce element type
auto x3{ 1, 2 }; // decltype(x3) is std::initializer_list<int>
auto x4 = { 3 }; // decltype(x4) is std::initializer_list<int>
auto x5{ 3 }; // decltype(x5) is int
```
—end example]

[Example 2:
```
const auto &i = expr;
The type of \( i \) is the deduced type of the parameter \( u \) in the call \( f(expr) \) of the following invented function template:
```template <class U> void f(const U& u);
```
—end example]

5 If the placeholder-type-specifier is of the form type-constraint_opt decltype(auto), \( T \) shall be the placeholder alone. The type deduced for \( T \) is determined as described in 9.2.9.5, as though \( E \) had been the operand of the decltype.

[Example 3:
```
int i;
int& f();
auto x2a(i); // decltype(x2a) is int
dectype(auto) x2d(i); // decltype(x2d) is int
auto x3a = i; // decltype(x3a) is int
dectype(auto) x3d = i; // decltype(x3d) is int
auto x4a = (i); // decltype(x4a) is int
dectype(auto) x4d = (i); // decltype(x4d) is int&
auto x5a = f(); // decltype(x5a) is int
dectype(auto) x5d = f(); // decltype(x5d) is int&&
auto x6a = { 1, 2 }; // decltype(x6a) is std::initializer_list<int>
dectype(auto) x6d = { 1, 2 }; // error: \{ 1, 2 \} is not an expression
auto *x7a = &i; // decltype(x7a) is int*
dectype(auto)*x7d = &i; // error: declared type is not plain decltype(auto)
```
—end example]

6 For a placeholder-type-specifier with a type-constraint, the immediately-declared constraint (13.2) of the type-constraint for the type deduced for the placeholder shall be satisfied.

9.2.9.7 Deduced class template specialization types [dcl.type.class.deduct]

1 If a placeholder for a deduced class type appears as a decl-specifier in the decl-specifier-seq of an initializing declaration (9.4) of a variable, the declared type of the variable shall be cv T, where \( T \) is the placeholder.

[Example 1:
```
template <class ...T> struct A {
    A(T...) {};
};
A x[29]{}; // error: no declarator operators allowed
const &A y{}; // error: no declarator operators allowed
```
—end example]
The placeholder is replaced by the return type of the function selected by overload resolution for class template
deduction (12.4.2.9). If the decl-specifier-seq is followed by an init-declarator-list or member-declarator-list
containing more than one declarator, the type that replaces the placeholder shall be the same in each
deduction.

A placeholder for a deduced class type can also be used in the type-specifier-seq in the new-type-id or
type-id of a new-expression (7.6.2.8), as the simple-type-specifier in an explicit type conversion (functional
notation) (7.6.1.4), or as the type-specifier in the parameter-declaration of a template-parameter (13.2). A
placeholder for a deduced class type shall not appear in any other context.

[Example 2:]

```cpp
template<class T> struct container {
  container(T t) {}
  template<class Iter> container(Iter beg, Iter end);
};
template<class Iter>
container(Iter b, Iter e) -> container<typename std::iterator_traits<Iter>::value_type>;

std::vector<double> v = { /* ... */ }; // OK, deduces int for T
auto d = container(v.begin(), v.end()); // OK, deduces double for T
container e{5, 6}; // error: int is not an iterator
```

§ 9.3 Declarators

9.3.1 General

A declarator declares a single variable, function, or type, within a declaration. The init-declarator-list appearing
in a declaration is a comma-separated sequence of declarators, each of which can have an initializer.

```
init-declarator-list:
  init-declarator
  init-declarator-list, init-declarator

typedef declearator;
  declarator initializer_opt
  declarator requires-clause
```

The three components of a simple-declaration are the attributes (9.12), the specifiers (decl-specifier-seq; 9.2)
and the declarators (init-declarator-list). The specifiers indicate the type, storage class or other properties of
the entities being declared. The declarators specify the names of these entities and (optionally) modify the
type of the specifiers with operators such as * (pointer to) and () (function returning). Initial values can
also be specified in a declarator; initializers are discussed in 9.4 and 11.10.

Each init-declarator in a declaration is analyzed separately as if it was in a declaration by itself.

[Note 1: A declaration with several declarators is usually equivalent to the corresponding sequence of declarations
each with a single declarator. That is

T D1, D2, ... Dn;

is usually equivalent to

T D1; T D2; ... T Dn;

where T is a decl-specifier-seq and each Di is an init-declarator. One exception is when a name introduced by one
of the declarators hides a type name used by the decl-specifiers, so that when the same decl-specifiers are used in a
subsequent declaration, they do not have the same meaning, as in

```cpp
struct S { /* ... */ ;
S S, T; // declare two instances of struct S
```

which is not equivalent to

```cpp
struct S { /* ... */ ;
S S;
S T; // error
```

Another exception is when T is auto (9.2.9.6), for example:

```cpp
auto i = 1, j = 2.0; // error: deduced types for i and j do not match
```

§ 9.3.1 183
as opposed to

```c
auto i = 1;  // OK: i deduced to have type int
auto j = 2.0; // OK: j deduced to have type double
```

—end note

4 The optional requires-clause (13.1) in an init-declarator or member-declarator shall be present only if the declarator declares a templated function (9.3.4.6). When present after a declarator, the requires-clause is called the trailing requires-clause. The trailing requires-clause introduces the constraint-expression that results from interpreting its constraint-logical-or-expression as a constraint-expression.

[Example 1:

```c
void f1(int a) requires true; // error: non-templated function
template<typename T>
  auto f2(T a) -> bool requires true; // OK
template<typename T>
  auto f3(T a) requires true -> bool; // error: requires-clause precedes trailing-return-type
void (*pf)() requires true; // error: constraint on a variable
g(int (*)() requires true); // error: constraint on a parameter-declaration
auto* p = new void(*)(char) requires true; // error: not a function declaration
```

—end example]

5 Declarators have the syntax

*declarator:*
  ptr-declarator
  nopl-declarator parameters-and-qualifiers trailing-return-type

*ptr-declarator:*
  nopl-declarator
  ptr-operator ptr-declarator

*nopl-declarator:*
  declarator-id attribute-specifier-seq_opt
  nopl-declarator parameters-and-qualifiers
  nopl-declarator [ constant-expression_opt ] attribute-specifier-seq_opt
  ( ptr-declarator )

*parameters-and-qualifiers:*
  ( parameter-declaration-clause ) cv-qualifier-seq_opt
  ref-qualifier_opt noexcept-specifier_opt attribute-specifier-seq_opt

*trailing-return-type:*
  -> type-id

*ptr-operator:*
  * attribute-specifier-seq_opt cv-qualifier-seq_opt
  & attribute-specifier-seq_opt
  && attribute-specifier-seq_opt
  nested-name-specifier * attribute-specifier-seq_opt cv-qualifier-seq_opt

*cv-qualifier-seq:*
  cv-qualifier cv-qualifier-seq_opt

*cv-qualifier:*
  const
  volatile

*ref-qualifier:*
  &
  &&

*declarator-id:*
  ....opt id-expression

### 9.3.2 Type names

1 To specify type conversions explicitly, and as an argument of `sizeof`, `alignof`, `new`, or `typeid`, the name of a type shall be specified. This can be done with a type-id, which is syntactically a declaration for a variable or function of that type that omits the name of the entity.
type-id:
  type-specifier-seq abstract-declarator<opt>

defining-type-id:
  defining-type-specifier-seq abstract-declarator<opt>

abstract-declarator:
  ptr-abstract-declarator
  noptr-abstract-declarator<opt> parameters-and-qualifiers trailing-return-type
  abstract-pack-declarator

ptr-abstract-declarator:
  noptr-abstract-declarator
  ptr-operator ptr-abstract-declarator<opt>

noptr-abstract-declarator:
  noptr-abstract-declarator<opt> parameters-and-qualifiers
  noptr-abstract-declarator<opt> [ constant-expression<opt> ] attribute-specifier-seq<opt>
  ( ptr-abstract-declarator )

abstract-pack-declarator:
  noptr-abstract-pack-declarator
  ptr-operator abstract-pack-declarator

noptr-abstract-pack-declarator:
  noptr-abstract-pack-declarator parameters-and-qualifiers
  noptr-abstract-pack-declarator [ constant-expression<opt> ] attribute-specifier-seq<opt>

...

It is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier.

[Example 1:

```c
int i; // int i
int * pi; // int *pi
int *p[3]; // int *p[3]
int (*p3i)[3]; // int (*p3i)[3]
int *f(); // int *f()
int (*)(double) f; // int (*)(double)
```

name respectively the types “int”, “pointer to int”, “array of 3 pointers to int”, “pointer to array of 3 int”, “function of (no parameters) returning pointer to int”, and “pointer to a function of (double) returning int”. —end example]

A type can also be named (often more easily) by using a typedef (9.2.4).

### 9.3.3 Ambiguity resolution [dcl.ambig.res]

The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 8.9 can also occur in the context of a declaration. In that context, the choice is between a function declaration with a redundant set of parentheses around a parameter name and an object declaration with a function-style cast as the initializer. Just as for the ambiguities mentioned in 8.9, the resolution is to consider any construct that could possibly be a declaration a declaration.

[Note 1: A declaration can be explicitly disambiguated by adding parentheses around the argument. The ambiguity can be avoided by use of copy-initialization or list-initialization syntax, or by use of a non-function-style cast. —end note]

[Example 1:

```c
struct S {
  S(int);  // object declaration
};

void foo(double a) {
  S u(int(a));  // function declaration
  S x(int());   // function declaration
  S y((int(a))); // object declaration
  S y((int)a);  // object declaration
  S z = int(a); // object declaration
}
```
An ambiguity can arise from the similarity between a function-style cast and a type-id. The resolution is that any construct that could possibly be a type-id in its syntactic context shall be considered a type-id.

Example 2:

```
template <class T> struct X {};
template <int N> struct Y {};
X<int()> a;          // type-id
X<int()> b;          // expression (ill-formed)
Y<int()> c;          // type-id (ill-formed)
Y<int()> d;          // expression

void foo(signed char a) {
    sizeof(int());    // type-id (ill-formed)
    sizeof(int(a));   // expression
    sizeof(int(unsigned(a))); // type-id (ill-formed)

    (int())+1;        // type-id (ill-formed)
    (int(a))+1;       // expression
    (int(unsigned(a)))+1; // type-id (ill-formed)
}
```

Another ambiguity arises in a parameter-declaration-clause when a type-name is nested in parentheses. In this case, the choice is between the declaration of a parameter of type pointer to function and the declaration of a parameter with redundant parentheses around the declarator-id. The resolution is to consider the type-name as a simple-type-specifier rather than a declarator-id.

Example 3:

```
class C { };
void f(int(C)) { }    // void f(int(*fp)(C c)) { }  // not: void f(int C) { }

int g(C);

void foo() {
    f(1);           // error: cannot convert 1 to function pointer
    f(g);           // OK
}
```

For another example,

```
class C { };
void h(int *(C[10])); // void h(int *(fp)(C _parm[10]));  // not: void h(int *C[10]);
```

9.3.4 Meaning of declarators [dcl.meaning]

9.3.4.1 General [dcl.meaning.general]

A declarator contains exactly one declarator-id; it names the identifier that is declared. An unqualified-id occurring in a declarator-id shall be a simple identifier except for the declaration of some special functions (11.4.5, 11.4.8, 11.4.7, 12.6) and for the declaration of template specializations or partial specializations (13.9). When the declarator-id is qualified, the declaration shall refer to a previously declared member of the class or namespace to which the qualifier refers (or, in the case of a namespace, of an element of the inline namespace set of that namespace (9.8.2)) or to a specialization thereof; the member shall not merely have been introduced by a using-declaration in the scope of the class or namespace nominated by the nested-name-specifier of the declarator-id. The nested-name-specifier of a qualified declarator-id shall not begin with a decltype-specifier.

[Note 1: If the qualifier is the global :: scope resolution operator, the declarator-id refers to a name declared in the global namespace scope. — end note]

The optional attribute-specifier-seq following a declarator-id appertains to the entity that is declared.
A static, thread_local, extern, mutable, friend, inline, virtual, constexpr, or typedef specifier or an explicit-specifier applies directly to each declarator-id in an init-declarator-list or member-declarator-list; the type specified for each declarator-id depends on both the decl-specifier-seq and its declarator.

Thus, a declaration of a particular identifier has the form

\[ T \ D \]

where \( T \) is of the form attribute-specifier-seq \_opt decl-specifier-seq and \( D \) is a declarator. Following is a recursive procedure for determining the type specified for the contained declarator-id by such a declaration.

First, the decl-specifier-seq determines a type. In a declaration

\[ T \ D \]

the decl-specifier-seq \( T \) determines the type \( T \).

[Example 1: In the declaration

\[
\text{int unsigned } i; \\
\text{the type specifiers } \text{int unsigned} \text{ determine the type } \text{"unsigned int" (9.2.9.3). — end example]}

In a declaration attribute-specifier-seq \_opt \( T \ D \) where \( D \) is an unadorned identifier the type of this identifier is "\( T \)".

In a declaration \( T \ D \) where \( D \) has the form

\[
( D1 )
\]

the type of the contained declarator-id is the same as that of the contained declarator-id in the declaration

\[ T \ D1 \]

Parentheses do not alter the type of the embedded declarator-id, but they can alter the binding of complex declarators.

9.3.4.2 Pointers [decl.ptr]

In a declaration \( T \ D \) where \( D \) has the form

\[
* \text{attribute-specifier-seq}_\text{opt} \text{ cv-qualifier-seq}_\text{opt} D1
\]

and the type of the identifier in the declaration \( T \ D1 \) is "derived-declarator-type-list \( T \)"; then the type of the identifier of \( D \) is "derived-declarator-type-list cv-qualifier-seq pointer to \( T \)". The cv-qualifiers apply to the pointer and not to the object pointed to. Similarly, the optional attribute-specifier-seq (9.12.1) appertains to the pointer and not to the object pointed to.

[Example 1: The declarations

\[
\text{const int } ci = 10, *pc = &ci, *\text{const } cpc = pc, **ppc; \\
\text{int } i, *p, *\text{const } cp = &i;
\]

declare \( ci \), a constant integer; \( pc \), a pointer to a constant integer; \( cpc \), a constant pointer to a constant integer; \( ppc \), a pointer to a pointer to a constant integer; \( i \), an integer; \( p \), a pointer to integer; and \( cp \), a constant pointer to integer.

The value of \( ci \), \( cpc \), and \( cp \) cannot be changed after initialization. The value of \( pc \) can be changed, and so can the object pointed to by \( cp \). Examples of some correct operations are

\[
i = ci; \\
*i = ci; \\
*p++;
\]

Examples of ill-formed operations are

\[
\text{ci = 1; } \\
\text{ci++; } \\
*\text{pc = 2;} \\
\text{cp = &ci;} \\
\text{cpc++; } \\
\text{p = pc;} \\
\text{ppc = &p; }
\]

Each is unacceptable because it would either change the value of an object declared \( \text{const} \) or allow it to be changed through a cv-unqualified pointer later, for example:
*ppc = &ci;       // OK, but would make p point to ci because of previous error
*p = 5;          // clobber ci
—end example]  

3 See also 7.6.19 and 9.4.  

4 [Note 1: Forming a pointer to reference type is ill-formed; see 9.3.4.3. Forming a function pointer type is ill-formed if
the function type has cv-qualifiers or a ref-qualifier; see 9.3.4.6. Since the address of a bit-field (11.4.10) cannot be
taken, a pointer can never point to a bit-field. — end note]  

9.3.4.3 References [dcl.ref]  

1 In a declaration T D where D has either of the forms

& attribute-specifier-seq_opt D1
&k attribute-specifier-seq_opt D1

and the type of the identifier in the declaration T D1 is “derived-declarator-type-list T”, then the type of the
identifier of D is “derived-declarator-type-list reference to T”. The optional attribute-specifier-seq appertains
to the reference type. Cv-qualified references are ill-formed except when the cv-qualifiers are introduced
through the use of a typedef-name (9.2.4, 13.2) or decltype-specifier (9.2.9.5), in which case the cv-qualifiers
are ignored.

[Example 1:  

typedef int& A;
const A aref = 3; // error: lvalue reference to non-const initialized with rvalue
The type of aref is “lvalue reference to int”, not “lvalue reference to const int”. — end example]  
[Note 1: A reference can be thought of as a name of an object. — end note]  

A declarator that specifies the type “reference to cv void” is ill-formed.

2 A reference type that is declared using & is called an lvalue reference, and a reference type that is declared
using && is called an rvalue reference. Lvalue references and rvalue references are distinct types. Except
where explicitly noted, they are semantically equivalent and commonly referred to as references.

3 [Example 2:  

    void f(double& a) { a += 3.14; }
    // ...  
    double d = 0;
    f(d);  

declares a to be a reference parameter of f so the call f(d) will add 3.14 to d.

    int v[20];
    // ...  
    int& g(int i) { return v[i]; }  
    // ...  
    g(3) = 7;

declares the function g() to return a reference to an integer so g(3)=7 will assign 7 to the fourth element of the array
v. For another example,

    struct link {
        link* next;
    };

    link* first;

    void h(link*& p) { // p is a reference to pointer
        p->next = first;
        first = p;
        p = 0;
    }

    void k() {
        link* q = new link;
        h(q);
    }  

§ 9.3.4.3 188
declares p to be a reference to a pointer to link so h(q) will leave q with the value zero. See also 9.4.4. — end example]

4 It is unspecified whether or not a reference requires storage (6.7.5).

5 There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an initializer (9.4.4) except when the declaration contains an explicit extern specifier (9.2.2), is a class member (11.4) declaration within a class definition, or is the declaration of a parameter or a return type (9.3.4.6); see 6.2. A reference shall be initialized to refer to a valid object or function.

[Note 2: In particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the “object” obtained by indirection through a null pointer, which causes undefined behavior. As described in 11.4.10, a reference cannot be bound directly to a bit-field. — end note]

6 If a typedef-name (9.2.4, 13.2) or a decltype-specifier (9.2.9.5) denotes a type TR that is a reference to a type T, an attempt to create the type “lvalue reference to cv TR” creates the type “lvalue reference to T”, while an attempt to create the type “value reference to cv TR” creates the type TR.

[Note 3: This rule is known as reference collapsing. — end note]

[Example 3:

```c
int i;
typedef int& LRI;
typedef int&& RRI;

LRI& r1 = i; // r1 has the type int
const LRI& r2 = i; // r2 has the type int
const LRI&& r3 = i; // r3 has the type int

RRI& r4 = i; // r4 has the type int
RRI&& r5 = 5; // r5 has the type int

decltype(r2)& r6 = i; // r6 has the type int
decltype(r2)&& r7 = i; // r7 has the type int
```
—end example]

7 [Note 4: Forming a reference to function type is ill-formed if the function type has cv-qualifier s or a ref-qualifier; see 9.3.4.6. — end note]

9.3.4.4 Pointers to members [dcl.mpnt]

1 In a declaration T D where D has the form

```
nested-name-specifier * attribute-specifier-seq_opt cv-qualifier-seq_opt D1
```

and the nested-name-specifier denotes a class, and the type of the identifier in the declaration T D1 is “derived-declarator-type-list T”, then the type of the identifier of D is “derived-declarator-type-list cv-qualifier-seq pointer to member of class nested-name-specifier of type D”. The optional attribute-specifier-seq (9.12.1) appertains to the pointer-to-member.

2 [Example 1:

```c
struct X {
    void f(int);
    int a;
};

struct Y;

int X::* pm1 = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pnc;
```

declares pm1, pmf, pmd and pnc to be a pointer to a member of X of type int, a pointer to a member of X of type void(int), a pointer to a member of X of type double and a pointer to a member of Y of type char respectively. The declaration of pmd is well-formed even though X has no members of type double. Similarly, the declaration of pnc is well-formed even though Y is an incomplete type. pm1 and pmf can be used like this:

```c
X obj;
```
// ... 
obj.*pmi = 7;  // assign 7 to an integer member of obj
(obj.*pmf)(7);  // call a function member of obj with the argument 7
—end example

3 A pointer to member shall not point to a static member of a class (11.4.9), a member with reference type, or “cv void”.

4 [Note 1: See also 7.6.2 and 7.6.4. The type “pointer to member” is distinct from the type “pointer”, that is, a pointer to member is declared only by the pointer-to-member declarator syntax, and never by the pointer declarator syntax. There is no “reference-to-member” type in C++. — end note]

9.3.4.5 Arrays [dcl.array]

1 In a declaration T D where D has the form

\[ \text{T } \text{D } \text{[ constant-expression}_{\text{opt} } \text{ attribute-specifier-seq}_{\text{opt} } \]

and the type of the contained declarator-id in the declaration T D1 is “derived-declarator-type-list T”, the type of the declarator-id in D is “derived-declarator-type-list array of N T”. The constant-expression shall be a converted constant expression of type \text{std::size_t} (7.7). Its value N specifies the array bound, i.e., the number of elements in the array; N shall be greater than zero.

2 In a declaration T D where D has the form

\[ \text{T } \text{D } \text{[ ] attribute-specifier-seq}_{\text{opt} } \]

and the type of the contained declarator-id in the declaration T D1 is “derived-declarator-type-list T”, the type of the declarator-id in D is “derived-declarator-type-list array of unknown bound of T”, except as specified below.

3 A type of the form “array of N U” or “array of unknown bound of U” is an array type. The optional attribute-specifier-seq appertains to the array type.

4 U is called the array element type; this type shall not be a placeholder type (9.2.9.6), a reference type, a function type, an array of unknown bound, or cv void.

[Note 1: An array can be constructed from one of the fundamental types (except void), from a pointer, from a pointer to member, from a class, from an enumeration type, or from an array of known bound. — end note]

[Example 1]:

\[ \text{float } \text{fa}[17], \*\text{afp}[17]; \]
declares an array of float numbers and an array of pointers to float numbers. — end example]

5 Any type of the form “cv-qualifier-seq array of N U” is adjusted to “array of N cv-qualifier-seq U”, and similarly for “array of unknown bound of U”.

[Example 2]:

\[ \text{typedef int } \text{A}[5], \text{AA}[2][3]; \]
\[ \text{typedef const A CA; } \]
\[ \text{typedef const AA CAA; } \]

—end example]

[Note 2: An “array of N cv-qualifier-seq U” has cv-qualified type; see 6.8.4. — end note]

6 An object of type “array of N U” consists of a contiguously allocated non-empty set of N subobjects of type U, known as the elements of the array, and numbered 0 to N-1.

7 In addition to declarations in which an incomplete object type is allowed, an array bound may be omitted in some cases in the declaration of a function parameter (9.3.4.6). An array bound may also be omitted when an object (but not a non-static data member) of array type is initialized and the declarator is followed by an initializer (9.4, 11.4, 7.6.1.4, 7.6.2.8). In these cases, the array bound is calculated from the number of initial elements (say, N) supplied (9.4.2), and the type of the array is “array of N U”.

8 Furthermore, if there is a preceding declaration of the entity in the same scope in which the bound was specified, an omitted array bound is taken to be the same as in that earlier declaration, and similarly for the definition of a static data member of a class.

[Example 3]:

\[ \text{extern int x[10]; } \]
struct S {
    static int y[10];
};

int x[]; // OK: bound is 10
int S::y[]; // OK: bound is 10

void f() {
    extern int x[];
    int i = sizeof(x); // error: incomplete object type
    ...}

—end example

9 [Note 3: When several "array of" specifications are adjacent, a multidimensional array type is created; only the first of the constant expressions that specify the bounds of the arrays can be omitted.]

[Example 4:
    int x3d[3][5][7];
]
delares an array of three elements, each of which is an array of five elements, each of which is an array of seven integers. The overall array can be viewed as a three-dimensional array of integers, with rank $3 \times 5 \times 7$. Any of the expressions $x3d$, $x3d[1]$, $x3d[1][j]$, $x3d[1][j][k]$ can reasonably appear in an expression. The expression $x3d[1]$ is equivalent to $*(x3d + 1)$; in that expression, $x3d$ is subject to the array-to-pointer conversion (7.3.3) and is first converted to a pointer to a 2-dimensional array with rank $5 \times 7$ that points to the first element of $x3d$. Then $i$ is added, which on typical implementations involves multiplying $i$ by the length of the object to which the pointer points, which is sizeof(int) $\times 5 \times 7$. The result of the addition and indirection is an lvalue denoting the $i^{th}$ array element of $x3d$ (an array of five arrays of seven integers). If there is another subscript, the same argument applies again, so $x3d[1][j]$ is an lvalue denoting the $j^{th}$ array element of the $i^{th}$ array element of $x3d$ (an array of seven integers), and $x3d[1][j][k]$ is an lvalue denoting the $k^{th}$ array element of the $j^{th}$ array element of the $i^{th}$ array element of $x3d$ (an integer). —end example]

The first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations. —end note

10 [Note 4: Conversions affecting expressions of array type are described in 7.3.3. — end note]

11 [Note 5: The subscript operator can be overloaded for a class (12.6.5). For the operator’s built-in meaning, see 7.6.1.2. — end note]

9.3.4.6 Functions [decl.fct]

1 In a declaration $T D$ where $D$ has the form

\[
D1 ( \text{parameter-declaration-clause} ) \text{cv-qualifier-seq}_opt \text{ref-qualifier}_opt \text{noexcept-specifier}_opt \text{attribute-specifier-seq}_opt
\]

and the type of the contained declarator-id in the declaration $T D1$ is "derived-declarator-type-list $T$", the type of the declarator-id in $D$ is "derived-declarator-type-list noexcept opt function of parameter-type-list cv-qualifier-seq ref-qualifier opt returning $T$", where

(1.1) — the parameter-type-list is derived from the parameter-declaration-clause as described below and

(1.2) — the optional noexcept is present if and only if the exception specification (14.5) is non-throwing.

The optional attribute-specifier-seq appertains to the function type.

2 In a declaration $T D$ where $D$ has the form

\[
D1 ( \text{parameter-declaration-clause} ) \text{cv-qualifier-seq}_opt \text{ref-qualifier}_opt \text{noexcept-specifier}_opt \text{attribute-specifier-seq}_opt \text{trailing-return-type}
\]

and the type of the contained declarator-id in the declaration $T D1$ is "derived-declarator-type-list $T$", $T$ shall be the single type-specifier auto. The type of the declarator-id in $D$ is "derived-declarator-type-list noexcept opt function of parameter-type-list cv-qualifier-seq ref-qualifier opt returning $U$", where

(2.1) — the parameter-type-list is derived from the parameter-declaration-clause as described below,

(2.2) — $U$ is the type specified by the trailing-return-type, and

(2.3) — the optional noexcept is present if and only if the exception specification is non-throwing.

The optional attribute-specifier-seq appertains to the function type.
A type of either form is a *function type*.\(^{91}\)

\[\text{parameter-declaration-clause:}\\  \text{parameter-declaration-list}_{\text{opt}} \ldots_{\text{opt}}\\  \text{parameter-declaration-list} \ldots,\\  \text{parameter-declaration}\\\ \text{parameter-declaration-list}, \text{parameter-declaration}\\ \text{parameter-declaration:}\\  \text{attribute-specifier-seq}_{\text{opt}} \text{decl-specifier-seq} \text{declarator}\\  \text{attribute-specifier-seq}_{\text{opt}} \text{decl-specifier-seq} \text{declarator} = \text{initializer-clause}\\  \text{attribute-specifier-seq}_{\text{opt}} \text{decl-specifier-seq} \text{abstract-declarator}_{\text{opt}}\\  \text{attribute-specifier-seq}_{\text{opt}} \text{decl-specifier-seq} \text{abstract-declarator}_{\text{opt}} = \text{initializer-clause}\\\]

The optional *attribute-specifier-seq* in a *parameter-declaration* appertains to the parameter.

The *parameter-declaration-clause* determines the arguments that can be specified, and their processing, when the function is called.

*[Note 1: The *parameter-declaration-clause* is used to convert the arguments specified on the function call; see 7.6.1.3. — end note]*

If the *parameter-declaration-clause* is empty, the function takes no arguments. A parameter list consisting of a single unnamed parameter of non-dependent type *void* is equivalent to an empty parameter list. Except for this special case, a parameter shall not have type *cv void*. A parameter with volatile-qualified type is deprecated; see D.6. If the *parameter-declaration-clause* terminates with an ellipsis or a function parameter pack (13.7.4), the number of arguments shall be equal to or greater than the number of parameters that do not have a default argument and are not function parameter packs. Where syntactically correct and where “…” is not part of an *abstract-declarator*, “,…” is synonymous with “…”.

*[Example 1: The declaration\\  \text{int printf(const char*}, \ldots);\\ declares a function that can be called with varying numbers and types of arguments.\\  \text{printf("hellow world");}\\  \text{printf("a=%d b=%d", a, b);}\\ However, the first argument must be of a type that can be converted to a *const char*.* — end example]*

The type of a function is determined using the following rules. The type of each parameter (including function parameter packs) is determined from its own *decl-specifier-seq* and *declarator*. After determining the type of each parameter, any parameter of type “array of T” or of function type T is adjusted to be “pointer to T”. After producing the list of parameter types, any top-level *cv-qualifiers* modifying a parameter type are deleted when forming the function type. The resulting list of transformed parameter types and the presence or absence of the ellipsis or a function parameter pack is the function’s *parameter-type-list*.

*[Note 3: This transformation does not affect the types of the parameters. For example, *int(*)\(\text{const int p, decltype(p)\*)} and \text{int(*)(int, const int\*)} are identical types. — end note]*

A function type with a *cv-qualifier-seq* or a *ref-qualifier* (including a type named by *typedef-name* (9.2.4, 13.2)) shall appear only as:

(6.1) — the function type for a non-static member function,

(6.2) — the function type to which a pointer to member refers,

(6.3) — the top-level function type of a function typedef declaration or *alias-declaration*,

(6.4) — the *type-id* in the default argument of a *type-parameter* (13.2), or

(6.5) — the *type-id* of a *template-argument* for a *type-parameter* (13.4.2).

*[Example 2:\\  \text{typedef int FIC(int) const;}\\  \text{FIC f;} // error: does not declare a member function]*

(91) As indicated by syntax, *cv-qualifiers* are a significant component in function return types.
struct S {
  FIC f;       // OK
};
FIC S::*pm = &S::f;   // OK
—end example

The effect of a `cv-qualifier-seq` in a function declarator is not the same as adding `cv`-qualification on top of the function type. In the latter case, the `cv`-qualifiers are ignored.

[Note 1: A function type that has a `cv-qualifier-seq` is not a `cv`-qualified type; there are no `cv`-qualified function types. —end note]

[Example 3:}
typedef void F();
struct S {
  const F f;       // OK: equivalent to: void f();
};
—end example

The return type, the parameter-type-list, the `ref-qualifier`, the `cv-qualifier-seq`, and the exception specification, but not the default arguments (9.3.4.7) or the trailing `requires-clause` (9.3), are part of the function type.

[Note 5: Function types are checked during the assignments and initializations of pointers to functions, references to functions, and pointers to member functions. —end note]

[Example 4: The declaration]
int fseek(FILE*, long, int);
declares a function taking three arguments of the specified types, and returning `int` (9.2.9). —end example]

A single name can be used for several different functions in a single scope; this is function overloading (Clause 12). All declarations for a function shall have equivalent return types, parameter-type-lists, and `requires-clauses` (13.7.7.2).

Functions shall not have a return type of type array or function, although they may have a return type of type pointer or reference to such things. There shall be no arrays of functions, although there can be arrays of pointers to functions.

A `volatile-qualified` return type is deprecated; see D.6.

Types shall not be defined in return or parameter types.

A `typedef` of function type may be used to declare a function but shall not be used to define a function (9.5).

[Example 5:]
typedef void F();
F fv;       // OK: equivalent to void fv();
F fv {}     // error
void fv() {} // OK: definition of fv
—end example

An identifier can optionally be provided as a parameter name; if present in a function definition (9.5), it names a parameter.

[Note 6: In particular, parameter names are also optional in function definitions and names used for a parameter in different declarations and the definition of a function need not be the same. If a parameter name is present in a function declaration that is not a definition, it cannot be used outside of its function declarator because that is the extent of its potential scope (6.4.4). —end note]

[Example 6: The declaration]
int i,
  *pi,
  f(),
  *fpi(int),
  (*fpi)(const char*, const char*),
  (*fpi(int))(int);
declares an integer `i`, a pointer `pi` to an integer, a function `f` taking no arguments and returning an integer, a function `fpi` taking an integer argument and returning a pointer to an integer, a pointer `pif` to a function which takes two pointers to constant characters and returns an integer, a function `fpif` taking an integer argument and returning a
pointer to a function that takes an integer argument and returns an integer. It is especially useful to compare \texttt{fpi} and \texttt{pif}. The binding of \texttt{*fpi(int)} is \texttt{*(fpi(int))}, so the declaration suggests, and the same construction in an expression requires, the calling of a function \texttt{fpi}, and then using indirection through the (pointer) result to yield an integer. In the declarator \texttt{*(pif)(const char*, const char*)}, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function, which is then called.

—end example

[Note 7: Typedefs and trailing-return-types are sometimes convenient when the return type of a function is complex. For example, the function \texttt{fpif} above could have been declared

\begin{verbatim}
typedef int IFUNC(int);
IFUNC* fpif(int);
\end{verbatim}

A trailing-return-type is most useful for a type that would be more complicated to specify before the declarator-id:

\begin{verbatim}
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
\end{verbatim}

rather than

\begin{verbatim}
template <class T, class U> decltype((*(T*)0) + (*(U*)0)) add(T t, U u);
\end{verbatim}

—end note]

A non-template function is a function that is not a function template specialization.

[Note 8: A function template is not a function. —end note]

An abbreviated function template is a function declaration that has one or more generic parameter type placeholders (9.2.9.6). An abbreviated function template is equivalent to a function template (13.7.7) whose template-parameter-list includes one invented type template-parameter for each generic parameter type placeholder of the function declaration, in order of appearance. For a placeholder-type-specifier of the form \texttt{auto}, the invented parameter is an unconstrained type-parameter. For a placeholder-type-specifier of the form \texttt{type-constraint auto}, the invented parameter is a type-parameter with that type-constraint. The invented type template-parameter is a template parameter pack if the corresponding parameter-declaration declares a function parameter pack. If the placeholder contains \texttt{decltype(auto)}, the program is ill-formed. The adjusted function parameters of an abbreviated function template are derived from the parameter-declaration by replacing each occurrence of a placeholder with the name of the corresponding invented template-parameter.

[Example 7:

\begin{verbatim}
template<typename T> concept C1 = /*...*/;
template<typename T> concept C2 = /*...*/;
template<typename... Ts> concept C3 = /*...*/;

void g1(const C1 auto*, C2 auto&);
void g2(C1 auto&...);
void g3(C3 auto...);
void g4(C3 auto);
\end{verbatim}

These declarations are functionally equivalent (but not equivalent) to the following declarations.

\begin{verbatim}
template<C1 T, C2 U> void g1(const T*, U&);
template<C1... Ts> void g2(Ts&...);
template<C3... Ts> void g3(Ts...);
template<C3 T> void g4(T);
\end{verbatim}

Abbreviated function templates can be specialized like all function templates.

\begin{verbatim}
template<> void g1<int>(const int*, const double&); // OK, specialization of g1<int, const double>
\end{verbatim}

—end example]

An abbreviated function template can have a template-head. The invented template-parameters are appended to the template-parameter-list after the explicitly declared template-parameters.

[Example 8:

\begin{verbatim}
template<typename> concept C = /*...*/;

template <typename T, C U>
void g(T x, U y, C auto z);
\end{verbatim}

This is functionally equivalent to each of the following two declarations.

\begin{verbatim}
template<typename, int> concept C = /*...*/;

template <typename T, int C U>
void g(T x, U y, C auto z);
\end{verbatim}
template<typename T, C U, C W>
    void g(T x, U y, W z);

template<typename T, typename U, typename W>
    requires C<U> && C<W>
    void g(T x, U y, W z);

—end example

20 A function declaration at block scope shall not declare an abbreviated function template.

21 A declarator-id or abstract-declarator containing an ellipsis shall only be used in a parameter-declaration. When it is part of a parameter-declaration-clause, the parameter-declaration declares a function parameter pack (13.7.4). Otherwise, the parameter-declaration is part of a template-parameter-list and declares a template parameter pack; see 13.2. A function parameter pack is a pack expansion (13.7.4).

[Example 9:
    template<typename... T> void f(T (*...t)(int, int));
    int add(int, int);
    float subtract(int, int);
    void g()
    {
        f(add, subtract);
    }
    —end example]

22 There is a syntactic ambiguity when an ellipsis occurs at the end of a parameter-declaration-clause without a preceding comma. In this case, the ellipsis is parsed as part of the abstract-declarator if the type of the parameter either names a template parameter pack that has not been expanded or contains auto; otherwise, it is parsed as part of the parameter-declaration-clause.

9.3.4.7 Default arguments [dcl.fct.default]
1 If an initializer-clause is specified in a parameter-declaration this initializer-clause is used as a default argument.
[Note 1: Default arguments will be used in calls where trailing arguments are missing (7.6.1.3). — end note]

2 [Example 1: The declaration
    void point(int = 3, int = 4);
declares a function that can be called with zero, one, or two arguments of type int. It can be called in any of these ways:
    point(1,2); point(1); point();
The last two calls are equivalent to point(1,4) and point(3,4), respectively. — end example]

3 A default argument shall be specified only in the parameter-declaration-clause of a function declaration or lambda-declarator or in a template-parameter (13.2); in the latter case, the initializer-clause shall be an assignment-expression. A default argument shall not be specified for a template parameter pack or a function parameter pack. If it is specified in a parameter-declaration-clause, it shall not occur within a declarator or abstract-declarator of a parameter-declaration.

4 For non-template functions, default arguments can be added in later declarations of a function in the same scope. Declarations in different scopes have completely distinct sets of default arguments. That is, declarations in inner scopes do not acquire default arguments from declarations in outer scopes, and vice versa. In a given function declaration, each parameter subsequent to a parameter with a default argument shall have a default argument supplied in this or a previous declaration, unless the parameter was expanded from a parameter pack, or shall be a function parameter pack.
[Note 2: A default argument cannot be redefined by a later declaration (not even to the same value) (6.3). — end note]
[Example 2:]

92) One can explicitly disambiguate the parse either by introducing a comma (so the ellipsis will be parsed as part of the parameter-declaration-clause) or by introducing a name for the parameter (so the ellipsis will be parsed as part of the declarator-id).
93) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or typedef declarations.
void g(int = 0, ...); // OK, ellipsis is not a parameter so it can follow
void f(int, int);
void f(int, int = 7);
void h() {
    f(3); // OK, calls f(3, 7)
    void f(int = 1, int); // error: does not use default from surrounding scope
}
void m() {
    void f(int, int); // has no defaults
    f(4); // error: wrong number of arguments
    f(4); // OK
    f(4, 5); // OK, calls f(4, 5);
    f(int, int = 5); // error: cannot redefine, even to same value
}
void n() {
    f(6); // OK, calls f(6, 7)
}
template<class ... T> struct C {
    void f(int n = 0, T...);
};
C<int> c; // OK, instantiates declaration void C::f(int n = 0, int)
—end example

5 For a given inline function defined in different translation units, the accumulated sets of default arguments at
the end of the translation units shall be the same; no diagnostic is required. If a friend declaration specifies a
default argument expression, that declaration shall be a definition and shall be the only declaration of the
function or function template in the translation unit.

The default argument has the same semantic constraints as the initializer in a declaration of a variable of the
parameter type, using the copy-initialization semantics (9.4). The names in the default argument are bound,
and the semantic constraints are checked, at the point where the default argument appears. Name lookup
and checking of semantic constraints for default arguments in function templates and in member functions of
class templates are performed as described in 13.9.2.

[Example 3: In the following code, $g$ will be called with the value $f(2)$:

```cpp
int a = 1;
int f(int);
int g(int x = f(a)); // default argument: f(::a)

void h() {
    a = 2;
    { // Note: In member function declarations, names in default arguments are looked up as described in 6.5.2. Access
        int a = 3;
        g(); // g(f(::a))
    }
}
—end example]

[Note 3: In member function declarations, names in default arguments are looked up as described in 6.5.2. Access
checking applies to names in default arguments as described in 11.9. — end note]

6 Except for member functions of class templates, the default arguments in a member function definition that
appears outside of the class definition are added to the set of default arguments provided by the member
function declaration in the class definition; the program is ill-formed if a default constructor (11.4.5.2), copy or
move constructor (11.4.5.3), or copy or move assignment operator (11.4.6) is so declared. Default arguments
for a member function of a class template shall be specified on the initial declaration of the member function
within the class template.

[Example 4:

class C {
    void f(int i = 3);
    void g(int i, int j = 99);
};

—end example]
void C::f(int i = 3) {} // error: default argument already specified in class scope
void C::g(int i = 88, int j) {} // in this translation unit, C::g can be called with no argument
—end example

[Note 4: A local variable cannot be odr-used (6.3) in a default argument. — end note]

[Example 5:

```cpp
void f() {
  int i;
  extern void g(int x = i); // error
  extern void h(int x = sizeof(i)); // OK
  // ...
}
—end example]
```

8 [Note 5: The keyword this cannot appear in a default argument of a member function; see 7.5.2.

[Example 6:

```cpp
class A {
  void f(A* p = this) { } // error
};
—end example
—end note]
```

9 A default argument is evaluated each time the function is called with no argument for the corresponding parameter. A parameter shall not appear as a potentially-evaluated expression in a default argument. Parameters of a function declared before a default argument are in scope and can hide namespace and class member names.

[Example 7:

```cpp
int a;
int f(int a, int b = a); // error: parameter a used as default argument
typedef int I;
int g(float I, int b = I(2)); // error: parameter I found
int h(int a, int b = sizeof(a)); // OK, unevaluated operand
—end example]
```

A non-static member shall not appear in a default argument unless it appears as the id-expression of a class member access expression (7.6.1.5) or unless it is used to form a pointer to member (7.6.2.2).

[Example 8: The declaration of X::mem1() in the following example is ill-formed because no object is supplied for the non-static member X::a used as an initializer.

```cpp
int b;
class X {
  int a;
  int mem1(int i = a); // error: non-static member a used as default argument
  int mem2(int i = b); // OK; use X::b
  static int b;
};
```

The declaration of X::mem2() is meaningful, however, since no object is needed to access the static member X::b. Classes, objects, and members are described in Clause 11. — end example]

A default argument is not part of the type of a function.

[Example 9:

```cpp
int f(int = 0);

void h() {
  int j = f(1);
  int k = f(); // OK, means f(0)
}
int (*p1)(int) = &f;
int (*p2)() = &f; // error: type mismatch
```
When a declaration of a function is introduced by way of a using-declaration (9.9), any default argument information associated with the declaration is made known as well. If the function is redeclared thereafter in the namespace with additional default arguments, the additional arguments are also known at any point following the redeclaration where the using-declaration is in scope.

A virtual function call (11.7.3) uses the default arguments in the declaration of the virtual function determined by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides.

Example 10:

```cpp
struct A {
    virtual void f(int a = 7);
};
struct B : public A {
    void f(int a);
};
void m() {
    B* pb = new B;
    A* pa = pb;
    pa->f(); // OK, calls pa->B::f(7)
    pb->f(); // error: wrong number of arguments for B::f()
}
```

9.4 Initializers [dcl.init]

9.4.1 General [dcl.init.general]

The process of initialization described in 9.4 applies to all initializations regardless of syntactic context, including the initialization of a function parameter (7.6.1.3), the initialization of a return value (8.7.4), or when an initializer follows a declarator.

```
initializer:
    brace-or-equal-initializer
    ( expression-list )
brace-or-equal-initializer:
    = initializer-clause
    braced-init-list
initializer-clause:
    assignment-expression
    braced-init-list
braced-init-list:
    { initializer-list ,opt }
    { designated-initializer-list ,opt }
    { }
initializer-list:
    initializer-clause .opt
    initializer-list , initializer-clause .opt
designated-initializer-list:
    designated-initializer-clause
designated-initializer-clause:
    designated-initializer-clause , designated-initializer-clause
designator:
    . identifier
expr-or-braced-init-list:
    expression
    braced-init-list
```

Note 1: The rules in 9.4 apply even if the grammar permits only the brace-or-equal-initializer form of initializer in a given context. — end note]
Except for objects declared with the `constexpr` specifier, for which see 9.2.6, an `initializer` in the definition of a variable can consist of arbitrary expressions involving literals and previously declared variables and functions, regardless of the variable’s storage duration.

[Example 1:
\[
\begin{align*}
\text{int } & f(\text{int}); \\
\text{int } & a = 2; \\
\text{int } & b = f(a); \\
\text{int } & c(b);
\end{align*}
\]
— end example]

[Note 2: Default arguments are more restricted; see 9.3.4.7. — end note]

A declaration of a block-scope variable with external or internal linkage that has an `initializer` is ill-formed.

To zero-initialize an object or reference of type `T` means:

- (6.1) if `T` is a scalar type (6.8), the object is initialized to the value obtained by converting the integer literal 0 (zero) to `T`;
- (6.2) if `T` is a (possibly cv-qualified) non-union class type, its padding bits (6.8) are initialized to zero bits and each non-static data member, each non-virtual base class subobject, and, if the object is not a base class subobject, each virtual base class subobject is zero-initialized;
- (6.3) if `T` is a (possibly cv-qualified) union type, its padding bits (6.8) are initialized to zero bits and the object’s first non-static named data member is zero-initialized;
- (6.4) if `T` is an array type, each element is zero-initialized;
- (6.5) if `T` is a reference type, no initialization is performed.

To default-initialize an object of type `T` means:

- (7.1) if `T` is a (possibly cv-qualified) class type (Clause 11), constructors are considered. The applicable constructors are enumerated (12.4.2.4), and the best one for the `initializer` () is chosen through overload resolution (12.4). The constructor thus selected is called, with an empty argument list, to initialize the object.
- (7.2) if `T` is an array type, each element is default-initialized.
- (7.3) otherwise, no initialization is performed.

A class type `T` is `const-default-constructible` if default-initialization of `T` would invoke a user-provided constructor of `T` (not inherited from a base class) or if

- (7.4) each direct non-variant non-static data member `M` of `T` has a default member initializer or, if `M` is of class type `X` (or array thereof), `X` is const-default-constructible,
- (7.5) if `T` is a union with at least one non-static data member, exactly one variant member has a default member initializer,
- (7.6) if `T` is not a union, for each anonymous union member with at least one non-static data member (if any), exactly one non-static data member has a default member initializer, and
- (7.7) each potentially constructed base class of `T` is const-default-constructible.

If a program calls for the default-initialization of an object of a const-qualified type `T`, `T` shall be a const-default-constructible class type or array thereof.

To value-initialize an object of type `T` means:

- (8.1) if `T` is a (possibly cv-qualified) class type (Clause 11), then
  - (8.1.1) if `T` has either no default constructor (11.4.5.2) or a default constructor that is user-provided or deleted, then the object is default-initialized;
  - (8.1.2) otherwise, the object is zero-initialized and the semantic constraints for default-initialization are checked, and if `T` has a non-trivial default constructor, the object is default-initialized;
- (8.2) if `T` is an array type, then each element is value-initialized;
- (8.3) otherwise, the object is zero-initialized.

94) As specified in 7.3.12, converting an integer literal whose value is 0 to a pointer type results in a null pointer value.
A program that calls for default-initialization or value-initialization of an entity of reference type is ill-formed.

[Note 4: For every object of static storage duration, static initialization (6.9.3.2) is performed at program startup before any other initialization takes place. In some cases, additional initialization is done later. — end note]

If no initializer is specified for an object, the object is default-initialized.

An initializer for a static member is in the scope of the member’s class.

[Example 2:

```c
int a;

struct X {
    static int a;
    static int b;
};

int X::a = 1;
int X::b = a; // X::b = X::a
```

— end example]

If the entity being initialized does not have class type, the expression-list in a parenthesized initializer shall be a single expression.

The initialization that occurs in the = form of a brace-or-equal-initializer or condition (8.5), as well as in argument passing, function return, throwing an exception (14.2), handling an exception (14.4), and aggregate member initialization (9.4.2), is called copy-initialization.

[Note 5: Copy-initialization can invoke a move (11.4.5.3). — end note]

The initialization that occurs

- for an initializer that is a parenthesized expression-list or a braced-init-list,
- for a new-initializer (7.6.2.8),
- in a static_cast expression (7.6.1.9),
- in a functional notation type conversion (7.6.1.4), and
- in the braced-init-list form of a condition

is called direct-initialization.

The semantics of initializers are as follows. The destination type is the type of the object or reference being initialized and the source type is the type of the initializer expression. If the initializer is not a single (possibly parenthesized) expression, the source type is not defined.

- If the initializer is a (non-parenthesized) braced-init-list or is = braced-init-list, the object or reference is list-initialized (9.4.5).
- If the destination type is a reference type, see 9.4.4.
- If the destination type is an array of characters, an array of char8_t, an array of char16_t, an array of char32_t, or an array of wchar_t, and the initializer is a string-literal, see 9.4.3.
- If the initializer is (), the object is value-initialized.

[Note 6: Since () is not permitted by the syntax for initializer,

```c
X a();
```

is not the declaration of an object of class X, but the declaration of a function taking no argument and returning an X. The form () is permitted in certain other initialization contexts (7.6.2.8, 7.6.1.4, 11.10.3). — end note]

- Otherwise, if the destination type is an array, the object is initialized as follows. Let \(x_1, \ldots, x_k\) be the elements of the expression-list. If the destination type is an array of unknown bound, it is defined as having \(k\) elements. Let \(n\) denote the array size after this potential adjustment. If \(k\) is greater than \(n\), the program is ill-formed. Otherwise, the \(i\)th array element is copy-initialized with \(x_i\) for each \(1 \leq i \leq k\), and value-initialized for each \(k < i \leq n\). For each \(1 \leq i < j \leq n\), every value computation and side effect associated with the initialization of the \(i\)th element of the array is sequenced before those associated with the initialization of the \(j\)th element.

- Otherwise, if the destination type is a (possibly cv-qualified) class type:
If the initializer expression is a prvalue and the cv-unqualified version of the source type is the same
class as the class of the destination, the initializer expression is used to initialize the destination
object.

[Example 3: \( T \ x = T(T(T())) \); calls the \( T \) default constructor to initialize \( x \). — end example]

Otherwise, if the initialization is direct-initialization, or if it is copy-initialization where the
cv-unqualified version of the source type is the same class as, or a derived class of, the class of the
destination, constructors are considered. The applicable constructors are enumerated (12.4.2.4),
and the best one is chosen through overload resolution (12.4). Then:

— If overload resolution is successful, the selected constructor is called to initialize the object,
with the initializer expression or expression-list as its argument(s).

— Otherwise, if no constructor is viable, the destination type is an aggregate class, and the
initializer is a parenthesized expression-list, the object is initialized as follows. Let \( e_1, \ldots, e_n \)
be the elements of the aggregate (9.4.2). Let \( x_1, \ldots, x_k \) be the elements of the expression-list.
If \( k \) is greater than \( n \), the program is ill-formed. The element \( e_i \) is copy-initialized with \( x_i \) for
\( 1 \leq i \leq k \). The remaining elements are initialized with their default member initializers, if
any, and otherwise are value-initialized. For each \( 1 \leq i < j \leq n \), every value computation and
side effect associated with the initialization of \( e_i \) is sequenced before those associated with the
initialization of \( e_j \).

[Note 7: By contrast with direct-list-initialization, narrowing conversions (9.4.5) are permitted,
designators are not permitted, a temporary object bound to a reference does not have its lifetime
extended (6.7.7), and there is no brace elision.

[Example 4:
```c
struct A {
  int a;
  int&& r;
};

int f();
int n = 10;

A a1{1, f()}; // OK, lifetime is extended
A a2{1, f()}; // well-formed, but dangling reference
A a3{1.0, 1}; // error: narrowing conversion
A a4{1.0, 1}; // well-formed, but dangling reference
A a5{1.0, std::move(n)}; // OK
— end example]
— end note]

— Otherwise, the initialization is ill-formed.

— Otherwise (i.e., for the remaining copy-initialization cases), user-defined conversions that can
convert from the source type to the destination type or (when a conversion function is used) to a
derived class thereof are enumerated as described in 12.4.2.5, and the best one is chosen through
overload resolution (12.4). If the conversion cannot be done or is ambiguous, the initialization is
ill-formed. The function selected is called with the initializer expression as its argument; if the
function is a constructor, the call is a prvalue of the cv-unqualified version of the destination type
whose result object is initialized by the constructor. The call is used to direct-initialize, according
to the rules above, the object that is the destination of the copy-initialization.

— Otherwise, if the source type is a (possibly cv-qualified) class type, conversion functions are considered.
The applicable conversion functions are enumerated (12.4.2.6), and the best one is chosen through
overload resolution (12.4). The user-defined conversion so selected is called to convert the initializer
expression into the object being initialized. If the conversion cannot be done or is ambiguous, the
initialization is ill-formed.

— Otherwise, if the initialization is direct-initialization, the source type is std::nullptr_t, and the
destination type is bool, the initial value of the object being initialized is false.

— Otherwise, the initial value of the object being initialized is the (possibly converted) value of the
initializer expression. A standard conversion sequence (7.3) will be used, if necessary, to convert the

§ 9.4.1
initializer expression to the cv-unqualified version of the destination type; no user-defined conversions are considered. If the conversion cannot be done, the initialization is ill-formed. When initializing a bit-field with a value that it cannot represent, the resulting value of the bit-field is implementation-defined.

[Note 8: An expression of type "cv1 T" can initialize an object of type "cv2 T" independently of the cv-qualifiers cv1 and cv2.]

```cpp
int a;
const int b = a;
int c = b;
/* end note */
```

17 An initializer-clause followed by an ellipsis is a pack expansion (13.7.4).

18 If the initializer is a parenthesized expression-list, the expressions are evaluated in the order specified for function calls (7.6.1.3).

19 The same identifier shall not appear in multiple designators of a designated-initializer-list.

20 An object whose initialization has completed is deemed to be constructed, even if the object is of non-class type or no constructor of the object’s class is invoked for the initialization.

[Note 9: Such an object might have been value-initialized or initialized by aggregate initialization (9.4.2) or by an inherited constructor (11.10.4). — end note]

Destroying an object of class type invokes the destructor of the class. Destroying a scalar type has no effect other than ending the lifetime of the object (6.7.3). Destroying an array destroys each element in reverse subscript order.

21 A declaration that specifies the initialization of a variable, whether from an explicit initializer or by default-initialization, is called the initializing declaration of that variable.

[Note 10: In most cases this is the defining declaration (6.2) of the variable, but the initializing declaration of a non-inline static data member (11.4.9.3) might be the declaration within the class definition and not the definition at namespace scope. — end note]

### 9.4.2 Aggregates [dcl.init.aggr]

1 An aggregate is an array or a class (Clause 11) with

(1.1) — no user-declared or inherited constructors (11.4.5),

(1.2) — no private or protected direct non-static data members (11.9),

(1.3) — no virtual functions (11.7.3), and

(1.4) — no virtual, private, or protected base classes (11.7.2).

[Note 1: Aggregate initialization does not allow accessing protected and private base class’ members or constructors. — end note]

2 The elements of an aggregate are:

(2.1) — for an array, the array elements in increasing subscript order, or

(2.2) — for a class, the direct base classes in declaration order, followed by the direct non-static data members (11.4) that are not members of an anonymous union, in declaration order.

3 When an aggregate is initialized by an initializer list as specified in 9.4.5, the elements of the initializer list are taken as initializers for the elements of the aggregate. The explicitly initialized elements of the aggregate are determined as follows:

(3.1) — If the initializer list is a designated-initializer-list, the aggregate shall be of class type, the identifier in each designator shall name a direct non-static data member of the class, and the explicitly initialized elements of the aggregate are the elements that are, or contain, those members.

(3.2) — If the initializer list is an initializer-list, the explicitly initialized elements of the aggregate are the first n elements of the aggregate, where n is the number of elements in the initializer list.

(3.3) — Otherwise, the initializer list must be {}, and there are no explicitly initialized elements.

4 For each explicitly initialized element:

(4.1) — If the element is an anonymous union object and the initializer list is a designated-initializer-list, the anonymous union object is initialized by the designated-initializer-list \{ D \}, where D is the designated-
initializer-clause naming a member of the anonymous union object. There shall be only one such designated-initializer-clause.

[Example 1:

```c
struct C {
    union {
        int a;
        const char* p;
    }
    int x;
} c = { .a = 1, .x = 3 };
```

initializes c.a with 1 and c.x with 3. —end example]

(4.2) — Otherwise, the element is copy-initialized from the corresponding initializer-clause or is initialized with the brace-or-equal-initializer of the corresponding designated-initializer-clause. If that initializer is of the form assignment-expression or = assignment-expression and a narrowing conversion (9.4.5) is required to convert the expression, the program is ill-formed.

[Note 2: If an initializer is itself an initializer list, the element is list-initialized, which will result in a recursive application of the rules in this subclause if the element is an aggregate. — end note]

[Example 2:

```c
struct A {
    int x;
    struct B {
        int i;
        int j;
    } b;
} a = { 1, { 2, 3 } };  
```

initializes a.x with 1, a.b.i with 2, a.b.j with 3.

```c
struct base1 { int b1, b2 = 42; }
struct base2 {
    base2() {
        b3 = 42;
    }
    int b3;
};
struct derived : base1, base2 {
    int d;
};
```

```c
derived d1{{1, 2}, {}, 4};
derived d2{{}, {}, 4};
```

initializes d1.b1 with 1, d1.b2 with 2, d1.b3 with 42, d1.d with 4, and d2.b1 with 0, d2.b2 with 42, d2.b3 with 42, d2.d with 4. —end example]

For a non-union aggregate, each element that is not an explicitly initialized element is initialized as follows:

(5.1) — If the element has a default member initializer (11.4), the element is initialized from that initializer.

(5.2) — Otherwise, if the element is not a reference, the element is copy-initialized from an empty initializer list (9.4.5).

(5.3) — Otherwise, the program is ill-formed.

If the aggregate is a union and the initializer list is empty, then

(5.4) — if any variant member has a default member initializer, that member is initialized from its default member initializer;

(5.5) — otherwise, the first member of the union (if any) is copy-initialized from an empty initializer list.

[Example 3:

```c
struct S { int a; const char* b; int c; int d = b[0a]; }
S ss = { 1, "asdf" };
```

initializes ss.a with 1, ss.b with "asdf", ss.c with the value of an expression of the form int{} (that is, 0), and ss.d with the value of ss.b[ss.a] (that is, 's'), and in

§ 9.4.2
struct X { int i, j, k = 42; };
X a[] = { 1, 2, 3, 4, 5, 6 };
X b[2] = {{ 1, 2, 3 }, { 4, 5, 6 }};
a and b have the same value

struct A {
    string a;
    int b = 42;
    int c = -1;
};
A {.c=21} has the following steps:

— Initialize a with {}
— Initialize b with = 42
— Initialize c with = 21

The initializations of the elements of the aggregate are evaluated in the element order. That is, all value
computations and side effects associated with a given element are sequenced before those of any element that
follows it in order.

An aggregate that is a class can also be initialized with a single expression not enclosed in braces, as described
in 9.4.

The destructor for each element of class type is potentially invoked (11.4.7) from the context where the
aggregate initialization occurs.

[Note 3: This provision ensures that destructors can be called for fully-constructed subobjects in case an exception is
thrown (14.3). — end note]

An array of unknown bound initialized with a brace-enclosed initializer-list containing n initializer-clauses is
defined as having n elements (9.3.4.5).

[Example 4:
int x[] = { 1, 3, 5 };
declares and initializes x as a one-dimensional array that has three elements since no size was specified and there are
three initializers. — end example]

An array of unknown bound shall not be initialized with an empty braced-init-list {}.

[Note 4: A default member initializer does not determine the bound for a member array of unknown bound. Since
the default member initializer is ignored if a suitable mem-initializer is present (11.10.3), the default member initializer
is not considered to initialize the array of unknown bound.

[Example 5:
struct S {
    int y[] = { 0 }; // error: non-static data member of incomplete type
};
— end example]
— end note]

[Note 5: Static data members, non-static data members of anonymous union members, and unnamed bit-fields are
not considered elements of the aggregate.

[Example 6:
struct A {
    int i;
    static int s;
    int j;
    int :17;
    int k;
} a = { 1, 2, 3 };
Here, the second initializer 2 initializes a.j and not the static data member A::s, and the third initializer 3 initializes
a.k and not the unnamed bit-field before it. — end example]

95) The syntax provides for empty braced-init-lists, but nonetheless C++ does not have zero length arrays.

§ 9.4.2
An initializer-list is ill-formed if the number of initializer-clauses exceeds the number of elements of the aggregate.

[Example 7:]
```c
char cv[4] = { 'a', 's', 'd', 'f', 0 }; // error
```

is ill-formed. —end example]

If a member has a default member initializer and a potentially-evaluated subexpression thereof is an aggregate initialization that would use that default member initializer, the program is ill-formed.

[Example 8:]
```c
struct A;
extern A a;
struct A {
    const A& a1 { A{a,a} }; // OK
    const A& a2 { A{} }; // error
};
A a{a,a}; // OK

struct B {
    int m = B{}.m;
    // error
};

A a{a,a}; // OK
struct B {
    int n = B{}.n;
};

// OK
```

—end example]

If an aggregate class C contains a subaggregate element e with no elements, the initializer-clause for e shall not be omitted from an initializer-list for an object of type C unless the initializer-clauses for all elements of C following e are also omitted.

[Example 9:]
```c
struct S { } s;
struct A {
    S s1;
    int i1;
    S s2;
    int i2;
    S s3;
    int i3;
} a = {
    { }, // Required initialization
    0,
    s, // Required initialization
    0
}; // Initialization not required for A::s3 because A::i3 is also not initialized
```

—end example]

When initializing a multi-dimensional array, the initializer-clauses initialize the elements with the last (rightmost) index of the array varying the fastest (9.3.4.5).

[Example 10:]
```c
int x[2][2] = { 3, 1, 4, 2 }; // Required initialization
initializes x[0][0] to 3, x[0][1] to 1, x[1][0] to 4, and x[1][1] to 2. On the other hand,
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
}; // Initialization not required for A::s3 because A::i3 is also not initialized
```

initializes the first column of y (regarded as a two-dimensional array) and leaves the rest zero. —end example]

Braces can be elided in an initializer-list as follows. If the initializer-list begins with a left brace, then the succeeding comma-separated list of initializer-clauses initializes the elements of a subaggregate; it is erroneous for there to be more initializer-clauses than elements. If, however, the initializer-list for a subaggregate does not begin with a left brace, then only enough initializer-clauses from the list are taken to initialize the elements of the subaggregate; any remaining initializer-clauses are left to initialize the next element of the aggregate of which the current subaggregate is an element.
Example 11:

```c
float y[4][3] = {
    1, 3, 5,
    2, 4, 6,
    3, 5, 7,
};
```

is a completely-braced initialization: 1, 3, and 5 initialize the first row of the array y[0], namely y[0][0], y[0][1], and y[0][2]. Likewise the next two lines initialize y[1] and y[2]. The initializer ends early and therefore y[3]'s elements are initialized as if explicitly initialized with an expression of the form float(), that is, are initialized with 0.0. In the following example, braces in the initializer-list are elided; however the initializer-list has the same effect as the completely-braced initializer-list of the above example,

```c
float y[4][3] = {1, 3, 5, 2, 4, 6, 3, 5, 7};
```

The initializer for y begins with a left brace, but the one for y[0] does not, therefore three elements from the list are used. Likewise the next three are taken successively for y[1] and y[2]. — end example

16 All implicit type conversions (7.3) are considered when initializing the element with an assignment-expression. If the assignment-expression can initialize an element, the element is initialized. Otherwise, if the element is itself a subaggregate, brace elision is assumed and the assignment-expression is considered for the initialization of the first element of the subaggregate.

[Note 6: As specified above, brace elision cannot apply to subaggregates with no elements; an initializer-clause for the entire subobject is required. — end note]

Example 12:

```c
struct A {
    int i;
    operator int();
};
struct B {
    A a1, a2;
    int z;
};
A a;
B b = { 4, a, a };
```

Braces are elided around the initializer-clause for b.a1.i. b.a1.i is initialized with 4, b.a2 is initialized with a, b.z is initialized with whatever a.operator int() returns. — end example

17 [Note 7: An aggregate array or an aggregate class can contain elements of a class type with a user-declared constructor (11.4.5). Initialization of these aggregate objects is described in 11.10.2. — end note]

18 [Note 8: Whether the initialization of aggregates with static storage duration is static or dynamic is specified in 6.9.3.2, 6.9.3.3, and 8.8. — end note]

19 When a union is initialized with an initializer list, there shall not be more than one explicitly initialized element.

Example 13:

```c
union u { int a; const char* b; };
u a = { 1 };
u b = a;
u c = 1; // error
u d = { 0, "asdf" }; // error
u e = { "asdf" }; // error
u f = {.b = "asdf" }; // error
u g = {.a = 1, .b = "asdf" }; // error
```

— end example

20 [Note 9: As described above, the braces around the initializer-clause for a union member can be omitted if the union is a member of another aggregate. — end note]

### 9.4.3 Character arrays

An array of ordinary character type (6.8.2), char8_t array, char16_t array, char32_t array, or wchar_t array can be initialized by an ordinary string literal, UTF-8 string literal, UTF-16 string literal, UTF-32 string
literal, or wide string literal, respectively, or by an appropriately-typed string-literal enclosed in braces (5.13.5). Successive characters of the value of the string-literal initialize the elements of the array.

**Example 1:**
```c
char msg[] = "Syntax error on line %s\n";
```
shows a character array whose members are initialized with a string-literal. Note that because ‘\n’ is a single character and because a trailing ‘\0’ is appended, sizeof(msg) is 25. — end example]

2 There shall not be more initializers than there are array elements.

**Example 2:**
```c
char cv[4] = "asdf";  // error
```
is ill-formed since there is no space for the implied trailing ‘\0’. — end example]

3 If there are fewer initializers than there are array elements, each element not explicitly initialized shall be zero-initialized (9.4).

### 9.4.4 References [dcl.init.ref]

1 A variable whose declared type is “reference to T” (9.3.4.3) shall be initialized.

**Example 1:**
```c
int g(int) noexcept;
void f() {
    int i;
    int& r = i;  // r refers to i
    r = 1;      // the value of i becomes 1
    int* p = &r;  // p points to i
    int& rr = r;  // rr refers to what r refers to, that is, to i
    int (&rg)(int) = g;  // rg refers to the function g
    rg(i);         // calls function g
    int a[3];
    int (&ra)[3] = a;  // ra refers to the array a
    ra[1] = i;  // modifies a[1]
}
```
— end example]

2 A reference cannot be changed to refer to another object after initialization.

[Note 1: Assignment to a reference assigns to the object referred to by the reference (7.6.19). — end note]

Argument passing (7.6.1.3) and function value return (8.7.4) are initializations.

3 The initializer can be omitted for a reference only in a parameter declaration (9.3.4.6), in the declaration of a function return type, in the declaration of a class member within its class definition (11.4), and where the extern specifier is explicitly used.

**Example 2:**
```c
int& r1;  // error: initializer missing
extern int& r2;  // OK
```
— end example]

4 Given types “cv1 T1” and “cv2 T2”, “cv1 T1” is reference-related to “cv2 T2” if T1 is similar (7.3.6) to T2, or T1 is a base class of T2. “cv1 T1” is reference-compatible with “cv2 T2” if a prvalue of type “pointer to cv1 T1” can be converted to the type “pointer to cv1 T1” via a standard conversion sequence (7.3). In all cases where the reference-compatible relationship of two types is used to establish the validity of a reference binding and the standard conversion sequence would be ill-formed, a program that necessitates such a binding is ill-formed.

5 A reference to type “cv1 T1” is initialized by an expression of type “cv2 T2” as follows:

- (5.1) — If the reference is an lvalue reference and the initializer expression

- (5.1.1) — is an lvalue (but is not a bit-field), and “cv1 T1” is reference-compatible with “cv2 T2”, or

- (5.1.2) — has a class type (i.e., T2 is a class type), where T1 is not reference-related to T2, and can be converted to an lvalue of type “cv3 T3”, where “cv1 T1” is reference-compatible with “cv3 T3”

96) This requires a conversion function (11.4.8.3) returning a reference type.
(this conversion is selected by enumerating the applicable conversion functions (12.4.2.7) and choosing the best one through overload resolution (12.4)),

then the reference is bound to the initializer expression lvalue in the first case and to the lvalue result of the conversion in the second case (or, in either case, to the appropriate base class subobject of the object).

[Note 2: The usual lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done. — end note]

[Example 3:

double d = 2.0;
double& rd = d; // rd refers to d
const double& rcd = d; // rcd refers to d

struct A {};
struct B : A ( operator int&() ); b;
A& ra = b; // ra refers to A subobject in b
const & rca = b; // rca refers to A subobject in b
int& ir = B(); // ir refers to the result of B::operator int&

— end example]

(5.2) — Otherwise, if the reference is an lvalue reference to a type that is not const-qualified or is volatile-qualified, the program is ill-formed.

[Example 4:

double& rd2 = 2.0; // error: not an lvalue and reference not const
int i = 2;
double& rd3 = i; // error: type mismatch and reference not const

— end example]

(5.3) — Otherwise, if the initializer expression

(5.3.1) — is an rvalue (but not a bit-field) or function lvalue and “cv1 T1” is reference-compatible with “cv2 T2”, or

(5.3.2) — has a class type (i.e., T2 is a class type), where T1 is not reference-related to T2, and can be converted to an rvalue or function value of type “cv3 T3”, where “cv1 T1” is reference-compatible with “cv3 T3” (see 12.4.2.7),

then the value of the initializer expression in the first case and the result of the conversion in the second case is called the converted initializer. If the converted initializer is a prvalue, its type T4 is adjusted to type “cv1 T4” (7.3.6) and the temporary materialization conversion (7.3.5) is applied. In any case, the reference is bound to the resulting glvalue (or to an appropriate base class subobject).

[Example 5:

struct A {};
struct B : A ( ) b;
extern B f();
const & rca2 = f(); // bound to the A subobject of the B rvalue.
A& rra = f(); // same as above
struct X {
operator B();
operator int&();
} x;
const & r = x; // bound to the A subobject of the result of the conversion
int i2 = 42;
int& rr = static_cast<int&>(i2); // bound directly to i2
E& rrb = x; // bound directly to the result of operator B

— end example]

(5.4) — Otherwise:

(5.4.1) — If T1 or T2 is a class type and T1 is not reference-related to T2, user-defined conversions are considered using the rules for copy-initialization of an object of type “cv1 T1” by user-defined conversion (9.4, 12.4.2.5, 12.4.2.6); the program is ill-formed if the corresponding non-reference
copy-initialization would be ill-formed. The result of the call to the conversion function, as described for the non-reference copy-initialization, is then used to direct-initialize the reference. For this direct-initialization, user-defined conversions are not considered.

— Otherwise, the initializer expression is implicitly converted to a prvalue of type “cv1 T1”. The temporary materialization conversion is applied and the reference is bound to the result.

If T1 is reference-related to T2:

— cv1 shall be the same cv-qualification as, or greater cv-qualification than, cv2; and

— if the reference is an rvalue reference, the initializer expression shall not be an lvalue.

[Example 6:

```cpp
struct Banana { }
struct Enigma { operator const Banana(); }
struct Alaska { operator Banana&(); }

void enigmatic() {

typedef const Banana ConstBanana;
Banana &&banana1 = ConstBanana();     // error
Banana &&banana2 = Enigma();          // error
Banana &&banana3 = Alaska();          // error
}

const double& rcd2 = 2;           // rcd2 refers to temporary with value 2.0
double&& rrd = 2;                 // rrd refers to temporary with value 2.0
const volatile int cvi = 1;
const int& r2 = cvi;              // error: cv-qualifier dropped
struct & { operator volatile int&(); } a;
const int& r3 = a;                // error: cv-qualifier dropped
                                           // from result of conversion function
double d2 = 1.0;
double&& rrd2 = d2;               // error: initializer is lvalue of related type
struct X { operator int&(); };
int&& rri2 = X();                 // error: result of conversion function is lvalue of related type
int i3 = 2;
double&& rrd3 = i3;              // rrd3 refers to temporary with value 2.0
```

—end example]

In all cases except the last (i.e., implicitly converting the initializer expression to the referenced type), the reference is said to bind directly to the initializer expression.

6 [Note 3: 6.7.7 describes the lifetime of temporaries bound to references. —end note]

### 9.4.5 List-initialization

List-initialization is initialization of an object or reference from a braced-init-list. Such an initializer is called an initializer list, and the comma-separated initializer-clauses of the initializer-list or designated-initializer-clauses of the designated-initializer-list are called the elements of the initializer list. An initializer list may be empty. List-initialization can occur in direct-initialization or copy-initialization contexts; list-initialization in a direct-initialization context is called direct-list-initialization and list-initialization in a copy-initialization context is called copy-list-initialization.

1 List-initialization can be used

1. as the initializer in a variable definition (9.4)
2. as the initializer in a new-expression (7.6.2.8)
3. in a return statement (8.7.4)
4. as a for-range-initializer (8.6)
5. as a function argument (7.6.1.3)
6. as a subscript (7.6.1.2)
7. as an argument to a constructor invocation (9.4, 7.6.1.4)
8. as an initializer for a non-static data member (11.4)
9. in a mem-initializer (11.10.3)
§ 9.4.5 210
N4868

© ISO/IEC

(3.7)

— Otherwise, if T is a class type, constructors are considered. The applicable constructors are enumerated
and the best one is chosen through overload resolution (12.4, 12.4.2.8). If a narrowing conversion (see
below) is required to convert any of the arguments, the program is ill-formed.
[Example 4 :
struct S {
S(std::initializer_list<double>);
S(std::initializer_list<int>);
S();
// ...
};
S s1 = { 1.0, 2.0, 3.0 };
S s2 = { 1, 2, 3 };
S s3 = { };

// #1
// #2
// #3
// invoke #1
// invoke #2
// invoke #3

— end example]
[Example 5 :
struct Map {
Map(std::initializer_list<std::pair<std::string,int>>);
};
Map ship = {{"Sophie",14}, {"Surprise",28}};
— end example]
[Example 6 :
struct S {
// no initializer-list constructors
S(int, double, double);
S();
// ...
};
S s1 = { 1, 2, 3.0 };
S s2 { 1.0, 2, 3 };
S s3 { };

// #1
// #2
// OK: invoke #1
// error: narrowing
// OK: invoke #2

— end example]
(3.8)

— Otherwise, if T is an enumeration with a fixed underlying type (9.7.1) U, the initializer-list has a single
element v, v can be implicitly converted to U, and the initialization is direct-list-initialization, the object
is initialized with the value T(v) (7.6.1.4); if a narrowing conversion is required to convert v to U, the
program is ill-formed.
[Example 7 :
enum
byte
byte
byte
byte

byte : unsigned char { };
b { 42 };
c = { 42 };
d = byte{ 42 };
e { -1 };

//
//
//
//

OK
error
OK; same value as b
error

struct A { byte b; };
A a1 = { { 42 } };
A a2 = { byte{ 42 } };

// error
// OK

void f(byte);
f({ 42 });

// error

enum class Handle : uint32_t { Invalid = 0 };
Handle h { 42 };
// OK
— end example]
(3.9)

— Otherwise, if the initializer list has a single element of type E and either T is not a reference type or its
referenced type is reference-related to E, the object or reference is initialized from that element (by
copy-initialization for copy-list-initialization, or by direct-initialization for direct-list-initialization); if a
narrowing conversion (see below) is required to convert the element to T, the program is ill-formed.
[Example 8 :

§ 9.4.5

211


int x1 {2}; // OK
int x2 {2.0}; // error: narrowing

— end example

— Otherwise, if T is a reference type, a prvalue is generated. The prvalue initializes its result object by copy-list-initialization. The prvalue is then used to direct-initialize the reference. The type of the temporary is the type referenced by T, unless T is “reference to array of unknown bound of U”, in which case the type of the temporary is the type of x in the declaration U x[] H, where H is the initializer list.

[Note 3: As usual, the binding will fail and the program is ill-formed if the reference type is an lvalue reference to a non-const type. — end note]

[Example 9]:

struct S {
 S(std::initializer_list<double>); // #1
 S(const std::string&); // #2
 // ...
};
const S& r1 = { 1, 2, 3.0 }; // OK: invoke #1
const S& r2 = "Spinach"; // OK: invoke #2
S& r3 = { 1, 2, 3 }; // error: initializer is not an lvalue
const int& i1 = { 1 }; // OK
const int& i2 = { 1.1 }; // error: narrowing
const int (&iar)[2] = { 1, 2 }; // OK: iar is bound to temporary array

struct A { } a;
struct B { explicit B(const A&); }
const B& b2(a); // error: cannot copy-list-initialize B temporary from A

— end example

— Otherwise, if the initializer list has no elements, the object is value-initialized.

[Example 10]:

int** pp {}; // initialized to null pointer
— end example

— Otherwise, the program is ill-formed.

[Example 11]:

struct A { int i; int j; }
A a1 { 1, 2 }; // aggregate initialization
A a2 { 1.2 }; // error: narrowing
struct B {
 B(std::initializer_list<int>);
};
B b1 { 1, 2 }; // creates initializer_list<int> and calls constructor
B b2 { 1, 2.0 }; // error: narrowing
struct C {
 C(int i, double j);
};
C c1 = { 1, 2.2 }; // calls constructor with arguments (1, 2.2)
C c2 = { 1.1, 2 }; // error: narrowing

int j { 1 }; // initialize to 1
int k {}; // initialize to 0
— end example

4 Within the initializer-list of a braced-init-list, the initializer-clauses, including any that result from pack expansions (13.7.4), are evaluated in the order in which they appear. That is, every value computation and side effect associated with a given initializer-clause is sequenced before every value computation and side effect associated with any initializer-clause that follows it in the comma-separated list of the initializer-list.

[Note 4: This evaluation ordering holds regardless of the semantics of the initialization; for example, it applies when the elements of the initializer-list are interpreted as arguments of a constructor call, even though ordinarily there are no sequencing constraints on the arguments of a call. — end note]
An object of type `std::initializer_list<E>` is constructed from an initializer list as if the implementation generated and materialized (7.3.5) a prvalue of type “array of \( N \) const \( E \)”, where \( N \) is the number of elements in the initializer list. Each element of that array is copy-initialized with the corresponding element of the initializer list, and the `std::initializer_list<E>` object is constructed to refer to that array.

[Note 5: A constructor or conversion function selected for the copy is required to be accessible (11.9) in the context of the initializer list. — end note]

If a narrowing conversion is required to initialize any of the elements, the program is ill-formed.

[Example 12:]
```cpp
struct X {
    X(std::initializer_list<double> v);
};
X x{ 1,2,3 }; // x is not a constant expression
```

The initialization will be implemented in a way roughly equivalent to this:

```cpp
const double __a[3] = {double{1}, double{2}, double{3}};
X x(std::initializer_list<double>(__a, __a+3));
```

assuming that the implementation can construct an `initializer_list` object with a pair of pointers. — end example]

The array has the same lifetime as any other temporary object (6.7.7), except that initializing an `initializer_list` object from the array extends the lifetime of the array exactly like binding a reference to a temporary.

[Example 13:]
```cpp
typedef std::complex<double> cmplx;
std::vector<cmplx> v1 = { 1, 2, 3 };

void f() {
    std::vector<cmplx> v2{ 1, 2, 3 };
    std::initializer_list<int> i3 = { 1, 2, 3 };
}

struct A {
    std::initializer_list<int> i4;
    A() : i4{ 1, 2, 3 } {} // ill-formed, would create a dangling reference
};
```

For `v1` and `v2`, the `initializer_list` object is a parameter in a function call, so the array created for \( \{ 1, 2, 3 \} \) has full-expression lifetime. For `i3`, the `initializer_list` object is a variable, so the array persists for the lifetime of the variable. For `i4`, the `initializer_list` object is initialized in the constructor’s `ctor-initializer` as if by binding a temporary array to a reference member, so the program is ill-formed (11.10.3). — end example]

[Note 6: The implementation is free to allocate the array in read-only memory if an explicit array with the same initializer could be so allocated. — end note]

A narrowing conversion is an implicit conversion

(7.1) — from a floating-point type to an integer type, or

(7.2) — from `long double` to `double` or `float`, or from `double` to `float`, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly), or

(7.3) — from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

(7.4) — from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression whose value after integral promotions will fit into the target type, or

(7.5) — from a pointer type or a pointer-to-member type to `bool`.

[Note 7: As indicated above, such conversions are not allowed at the top level in list-initializations. — end note]

[Example 14:]
```cpp
int x = 999;
const int y = 999;
```

// x is not a constant expression

§ 9.4.5 213
const int z = 99;
char c1 = x;  // OK, though it might narrow (in this case, it does narrow)
char c2(x);  // error: might narrow
char c3(y);  // error: narrows (assuming char is 8 bits)
char c4(z);  // OK: no narrowing needed
unsigned char uc1 = {5};  // OK: no narrowing needed
unsigned char uc2 = {-1};  // error: narrows
unsigned int ui1 = {-1};  // error: narrows
signed int si1 = (unsigned int)-1;  // error: narrows
int ii = (2.0);  // error: narrows
float f1 { x };  // error: might narrow
float f2 { 7 };  // OK: 7 can be exactly represented as a float
bool b = {"meow"};  // error: narrows
int f(int);
int a[] = { 2, f(2), f(2.0) };  // OK: the double-to-int conversion is not at the top level

--- end example

9.5 Function definitions

9.5.1 In general

Function definitions have the form

```
fraction-definition:  
  attribute-specifier-seq_opt decl-specifier-seq_opt declarator virt-specifier-seq_opt function-body
  attribute-specifier-seq_opt decl-specifier-seq_opt declarator requires-clause function-body

function-body:  
  ctor-initializer_opt compound-statement
  function-try-block = default ;
  = delete ;
```

Any informal reference to the body of a function should be interpreted as a reference to the non-terminal `function-body`. The optional `attribute-specifier-seq` in a `function-definition` appertains to the function. A `virt-specifier-seq` can be part of a `function-definition` only if it is a `member-declaration` (11.4).

In a `function-definition`, either `void declarator` ; or `declarator` ; shall be a well-formed function declaration as described in 9.3.4.6. A function shall be defined only in namespace or class scope. The type of a parameter or the return type for a function definition shall not be a (possibly cv-qualified) class type that is incomplete or abstract within the function body unless the function is deleted (9.5.3).

[Example 1: A simple example of a complete function definition is]

```
int max(int a, int b, int c) {
  int m = (a > b) ? a : b;
  return (m > c) ? m : c;
}
```

Here `int` is the `decl-specifier-seq; max(int a, int b, int c)` is the `declarator`; `/* ... */` is the `function-body`.  

--- end example

4 A `ctor-initializer` is used only in a constructor; see 11.4.5 and 11.10.

5 [Note 1: A cv-qualifier-seq affects the type of this in the body of a member function; see 7.5.2. — end note]

6 [Note 2: Unused parameters need not be named. For example,

```
void print(int a, int) {
  std::printf("a = %d\n",a);
}
```

— end note]

7 In the `function-body`, a `function-local predefined variable` denotes a block-scope object of static storage duration that is implicitly defined (see 6.4.3).

8 The `function-local predefined variable` `_func_` is defined as if a definition of the form

```
static const char _func_[] = "function-name";
```
had been provided, where \texttt{function-name} is an implementation-defined string. It is unspecified whether such a variable has an address distinct from that of any other object in the program.\footnote{Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (5.10). If a predefined variable is not odr-used (6.3), its string value need not be present in the program image.}

\textbf{Example 2:}

\begin{verbatim}
struct S {
    S() : s(__func__) { }    // OK
    const char* s;
};
void f(const char* s = __func__);    // error: __func__ is undeclared
\end{verbatim}

\section{Explicitly-defaulted functions} \[dcl.fct.def.default\]

A function definition whose \texttt{function-body} is of the form \texttt{= default;} is called an \textit{explicitly-defaulted} definition. A function that is explicitly defaulted shall

\begin{enumerate}
\item be a special member function or a comparison operator function (12.6.3), and
\item not have default arguments.
\end{enumerate}

The type \(T_1\) of an explicitly defaulted special member function \(F\) is allowed to differ from the type \(T_2\) it would have had if it were implicitly declared, as follows:

\begin{enumerate}
\item \(T_1\) and \(T_2\) may have differing \textit{ref-qualifiers};
\item \(T_1\) and \(T_2\) may have differing exception specifications; and
\item if \(T_2\) has a parameter of type \texttt{const C&}, the corresponding parameter of \(T_1\) may be of type \texttt{C&}.
\end{enumerate}

If \(T_1\) differs from \(T_2\) in any other way, then:

\begin{enumerate}
\item if \(F\) is an assignment operator, and the return type of \(T_1\) differs from the return type of \(T_2\) or \(T_1\)'s parameter type is not a reference, the program is ill-formed;
\item otherwise, if \(F\) is explicitly defaulted on its first declaration, it is defined as deleted;
\item otherwise, the program is ill-formed.
\end{enumerate}

An explicitly-defaulted function that is not defined as deleted may be declared \texttt{constexpr} or \texttt{consteval} only if it is \texttt{constexpr-compatible} (11.4.4, 11.11.1). A function explicitly defaulted on its first declaration is implicitly inline (9.2.8), and is implicitly constexpr (9.2.6) if it is constexpr-compatible.

\textbf{Example 1:}

\begin{verbatim}
struct S {
    constexpr S() = default;    // error: implicit S() is not constexpr
    S(int a = 0) = default;     // error: default argument
    void operator=(const S&) = default;    // error: non-matching return type
    ~S() noexcept(false) = default;    // OK, despite mismatched exception specification
private:
    int i;
    S(S&);                    // OK: private copy constructor
};
S::S(S&) = default;                 // OK: defines copy constructor

struct T {
    T();
    T(T&&) noexcept(false);
};
struct U {
    T t;
    U();
    U(U&&) noexcept = default;    // OK, calls std::terminate if T::T(T&&) throws
};
U u1;
U u2 = static_cast<U&&>(u1);        // OK, calls std::terminate if T::T(T&&) throws
\end{verbatim}

\footnote{\texttt{function-name} is an implementation-defined string. It is unspecified whether such a variable has an address distinct from that of any other object in the program.}
Explicitly-defaulted functions and implicitly-declared functions are collectively called defaulted functions, and the implementation shall provide implicit definitions for them (11.4.5, 11.4.7, 11.4.5.3, 11.4.6), including possibly defining them as deleted. A defaulted prospective destructor (11.4.7) that is not a destructor is defined as deleted. A defaulted special member function that is neither a prospective destructor nor an eligible special member function (11.4.4) is defined as deleted. A function is user-provided if it is user-declared and not explicitly defaulted or deleted on its first declaration. A user-provided explicitly-defaulted function (i.e., explicitly defaulted after its first declaration) is defined at the point where it is explicitly defaulted; if such a function is implicitly defined as deleted, the program is ill-formed.

[Note 1: Declaring a function as defaulted after its first declaration can provide efficient execution and concise definition while enabling a stable binary interface to an evolving code base. — end note]

[Example 2:

```cpp
struct trivial {
    trivial() = default;
    trivial(const trivial&) = default;
    trivial(trivial&&) = default;
    trivial& operator=(const trivial&) = default;
    trivial& operator=(trivial&&) = default;
    "-trivial() = default;
};
struct nontrivial1 {
    nontrivial1();
};
nontrivial1::nontrivial1() = default; // not first declaration
```

—end example]

§ 9.5.3 Deleted definitions
dcl.fct.def.delete

A function definition whose function-body is of the form = delete; is called a deleted definition. A function with a deleted definition is also called a deleted function.

A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed.

[Note 1: This includes calling the function implicitly or explicitly and forming a pointer or pointer-to-member to the function. It applies even for references in expressions that are not potentially-evaluated. If a function is overloaded, it is referenced only if the function is selected by overload resolution. The implicit odr-use (6.3) of a virtual function does not, by itself, constitute a reference. — end note]

[Example 1: One can prevent default initialization and initialization by non-doubles with

```cpp
struct onlydouble {
    onlydouble() = delete; // OK, but redundant
    template<class T>
    onlydouble(T) = delete;
    onlydouble(double);
};
```

—end example]

[Example 2: One can prevent use of a class in certain new-expressions by using deleted definitions of a user-declared operator new for that class.

```cpp
struct sometype {
    void* operator new(std::size_t) = delete;
    void* operator new[](std::size_t) = delete;
};
sometype* p = new sometype; // error: deleted class operator new
sometype* q = new sometype[3]; // error: deleted class operator new[]
```

—end example]

[Example 3: One can make a class uncopyable, i.e., move-only, by using deleted definitions of the copy constructor and copy assignment operator, and then providing defaulted definitions of the move constructor and move assignment operator.

```cpp
struct moveonly {
    moveonly() = default;
    moveonly(const moveonly&) = delete;
```
moveonly(moveonly&&) = default;
moveonly& operator=(const moveonly&) = delete;
moveonly& operator=(moveonly&&) = default;
~moveonly() = default;
);
moveonly* p;
moveonly q(*p); // error: deleted copy constructor
— end example

4 A deleted function is implicitly an inline function (9.2.8).

[Note 2: The one-definition rule (6.3) applies to deleted definitions. — end note]
A deleted definition of a function shall be the first declaration of the function or, for an explicit specialization of a function template, the first declaration of that specialization. An implicitly declared allocation or deallocation function (6.7.5.5) shall not be defined as deleted.

[Example 4]
struct sometype {
    sometype();
};
sometype::sometype() = delete; // error: not first declaration
— end example

9.5.4 Coroutine definitions

1 A function is a coroutine if its function-body encloses a coroutine-return-statement (8.7.5), an await-expression (7.6.2.4), or a yield-expression (7.6.17). The parameter-declaration-clause of the coroutine shall not terminate with an ellipsis that is not part of a parameter-declaration.

2 [Example 1]:

    task<int> f();
    
    task<void> g1() {
        int i = co_await f();
        std::cout << "f() => " << i << std::endl;
    }
    
    template <typename... Args>
task<void> g2(Args&&...){ // OK, ellipsis is a pack expansion
        int i = co_await f();
        std::cout << "f() => " << i << std::endl;
    }
    
    task<void> g3(int a, ...){ // error: variable parameter list not allowed
        int i = co_await f();
        std::cout << "f() => " << i << std::endl;
    }
— end example

3 The promise type of a coroutine is std::coroutine_traits<R, P_1, ..., P_n>::promise_type, where R is the return type of the function, and P_1...P_n are the sequence of types of the function parameters, preceded by the type of the implicit object parameter (12.4.2) if the coroutine is a non-static member function. The promise type shall be a class type.

4 In the following, p_i is an lvalue of type P_i, where p_1 denotes *this and p_{i+1} denotes the i_th function parameter for a non-static member function, and p_j denotes the j_th function parameter otherwise.

5 A coroutine behaves as if its function-body were replaced by:
```c++
{
    promise-type promise
    promise-constructor-arguments;
    try {
        co_await promise.initial_suspend();
        function-body
    } catch ( ... ) {
        if (!initial-await-resume-called)
            throw ;
        promise.unhandled_exception();
    }
    final-suspend:
    co_await promise.final_suspend();
}
```

where

- (5.1) the await-expression containing the call to `initial_suspend` is the initial suspend point, and
- (5.2) the await-expression containing the call to `final_suspend` is the final suspend point, and
- (5.3) `initial-await-resume-called` is initially `false` and is set to `true` immediately before the evaluation of the await-resume expression (7.6.2.4) of the initial suspend point, and
- (5.4) `promise-type` denotes the promise type, and
- (5.5) the object denoted by the exposition-only name `promise` is the promise object of the coroutine, and
- (5.6) the label denoted by the name `final-suspend` is defined for exposition only (8.7.5), and
- (5.7) `promise-constructor-arguments` is determined as follows: overload resolution is performed on a promise constructor call created by assembling an argument list with lvalues \( p_1 \ldots p_n \). If a viable constructor is found (12.4.3), then `promise-constructor-arguments` is \( (p_1, \ldots, p_n) \), otherwise `promise-constructor-arguments` is empty.

6 The unqualified-ids `return_void` and `return_value` are looked up in the scope of the promise type. If both are found, the program is ill-formed.

[Note 1: If the unqualified-id `return_void` is found, flowing off the end of a coroutine is equivalent to a `co_return` with no operand. Otherwise, flowing off the end of a coroutine results in undefined behavior (8.7.5). — end note]

7 The expression `promise.get_return_object()` is used to initialize the glvalue result or prvalue result object of a call to a coroutine. The call to `get_return_object` is sequenced before the call to `initial_suspend` and is invoked at most once.

8 A suspended coroutine can be resumed to continue execution by invoking a resumption member function (17.12.4.5) of a coroutine handle (17.12.4) that refers to the coroutine. The function that invoked a resumption member function is called the resumer. Invoking a resumption member function for a coroutine that is not suspended results in undefined behavior.

9 An implementation may need to allocate additional storage for a coroutine. This storage is known as the coroutine state and is obtained by calling a non-array allocation function (6.7.5.5.2). The allocation function’s name is looked up in the scope of the promise type. If this lookup fails, the allocation function’s name is looked up in the global scope. If the lookup finds an allocation function in the scope of the promise type, overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and has type `std::size_t`. The lvalues \( p_1 \ldots p_n \) are the succeeding arguments. If no viable function is found (12.4.3), overload resolution is performed again on a function call created by passing just the amount of space required as an argument of type `std::size_t`.

10 The unqualified-id `get_return_object_on_allocation_failure` is looked up in the scope of the promise type by class member access lookup (6.5.6). If any declarations are found, then the result of a call to an allocation function used to obtain storage for the coroutine state is assumed to return `nullptr` if it fails to obtain storage, and if a global allocation function is selected, the `::operator new(size_t, noexcept_t)` form is used. The allocation function used in this case shall have a non-throwing noexcept-specifier. If the allocation function returns `nullptr`, the coroutine returns control to the caller of the coroutine and the return value is obtained by a call to `T::get_return_object_on_allocation_failure()`, where `T` is the promise type.

[Example 2:}
```cpp
#include <iostream>
#include <coroutine>

// : operator new(size_t, nothrow_t) will be used if allocation is needed
struct generator {
    struct promise_type;
    using handle = std::coroutine_handle<promise_type>;
    struct promise_type {
        int current_value;
        static auto get_return_object_on_allocation_failure() { return generator(nullptr); }
        auto get_return_object() { return generator(handle::from_promise(*this)); }
        auto initial_suspend() { return std::suspend_always{}; }
        auto final_suspend() noexcept { return std::suspend_always{}; }
        void unhandled_exception() { std::terminate(); }
        void return_void() {}
        auto yield_value(int value) {
            current_value = value;
            return std::suspend_always{};
        }
    }
    bool move_next() { return coro ? (coro.resume(), !coro.done()) : false; }
    int current_value() { return coro.promise().current_value; }
    generator(generator const&) = delete;
    generator(generator && rhs) : coro(rhs.coro) { rhs.coro = nullptr; }
    ~generator() { if (coro) coro.destroy(); }
private:
    generator(handle h) : coro(h) {}
    handle coro;
};

generator f() { co_yield 1; co_yield 2; }

int main() {
    auto g = f();
    while (g.move_next()) std::cout << g.current_value() << std::endl;
}
```

11 The coroutine state is destroyed when control flows off the end of the coroutine or the destroy member
function (17.12.4.5) of a coroutine handle (17.12.4) that refers to the coroutine is invoked. In the latter case
objects with automatic storage duration that are in scope at the suspend point are destroyed in the reverse
order of the construction. The storage for the coroutine state is released by calling a non-array deallocation
function (6.7.5.5.3). If destroy is called for a coroutine that is not suspended, the program has undefined
behavior.

12 The deallocation function’s name is looked up in the scope of the promise type. If this lookup fails, the
deallocation function’s name is looked up in the global scope. If deallocation function lookup finds both a
usual deallocation function with only a pointer parameter and a usual deallocation function with both a
pointer parameter and a size parameter, then the selected deallocation function shall be the one with two
parameters. Otherwise, the selected deallocation function shall be the function with one parameter. If no
usual deallocation function is found, the program is ill-formed. The selected deallocation function shall be
called with the address of the block of storage to be reclaimed as its first argument. If a deallocation function
with a parameter of type std::size_t is used, the size of the block is passed as the corresponding argument.

13 When a coroutine is invoked, after initializing its parameters (7.6.1.3), a copy is created for each coroutine
parameter. For a parameter of type cv T, the copy is a variable of type cv T with automatic storage
duration that is direct-initialized from an xvalue of type T referring to the parameter.

[Note 2: An original parameter object is never a const or volatile object (6.8.4). — end note]
The initialization and destruction of each parameter copy occurs in the context of the called coroutine.
Initializations of parameter copies are sequenced before the call to the coroutine promise constructor and
indeterminately sequenced with respect to each other. The lifetime of parameter copies ends immediately
after the lifetime of the coroutine promise object ends.

[Note 3: If a coroutine has a parameter passed by reference, resuming the coroutine after the lifetime of the entity
referred to by that parameter has ended is likely to result in undefined behavior. — end note]
14 If the evaluation of the expression `promise.unhandled_exception()` exits via an exception, the coroutine is considered suspended at the final suspend point.

15 The expression `coAwait promise.final_suspend()` shall not be potentially-throwing (14.5).

### 9.6 Structured binding declarations

A structured binding declaration introduces the identifiers `v₀, v₁, v₂, ...` of the identifier-list as names (6.4.1) of structured bindings. Let `cv` denote the cv-qualifiers in the declarator-list and `S` consist of the storage-class-specifiers of the declarator-list (if any). A `cv` that includes `volatile` is deprecated; see D.6. First, a variable with a unique name `e` is introduced. If the assignment-expression in the `initializer` has array type `A` and no ref-qualifier is present, `e` is defined by

```cpp
attribute-specifier-seqopt S cv & e ;
```

and each element is copy-initialized or direct-initialized from the corresponding element of the assignment-expression as specified by the form of the `initializer`. Otherwise, `e` is defined as-if by

```cpp
attribute-specifier-seqopt decl-specifier-seq ref-qualifieropt e initializer ;
```

where the declaration is never interpreted as a function declaration and the parts of the declaration other than the declarator-id are taken from the corresponding structured binding declaration. The type of the id-expression `e` is called `E`.

[Note 1: `E` is never a reference type (7.2). — end note]

2 If the `initializer` refers to one of the names introduced by the structured binding declaration, the program is ill-formed.

3 If `E` is an array type with element type `T`, the number of elements in the identifier-list shall be equal to the number of elements of `E`. Each `vᵢ` is the name of an lvalue that refers to the element `i` of the array and whose type is `T`; the referenced type is `T`.

[Note 2: The top-level cv-qualifiers of `T` are `cv`. — end note]

[Example 1:

```cpp
auto f() -> int[k][2];
auto [ x, y ] = f(); // x and y refer to elements in a copy of the array return value
auto & [ xr, yr ] = f(); // xr and yr refer to elements in the array referred to by f's return value
```

— end example]

4 Otherwise, if the qualified-id `std::tuple_size<E>` names a complete class type with a member named `value`, the expression `std::tuple_size<E>::value` shall be a well-formed integral constant expression and the number of elements in the identifier-list shall be equal to the value of that expression. Let `i` be an index into the value of type `std::size_t` corresponding to `vᵢ`. The unqualified-id `get` is looked up in the scope of `E` by class member access lookup (6.5.6), and if that finds at least one declaration that is a function template whose first template parameter is a non-type parameter, the initializer is `e.get<i>()`. Otherwise, the initializer is `get<i>(e)`, where `get` is looked up in the associated namespaces (6.5.3). In either case, `get<i>` is interpreted as a template-id.

[Note 3: Ordinary unqualified lookup (6.5.2) is not performed. — end note]

In either case, `e` is an lvalue if the type of the entity `e` is an lvalue reference and an xvalue otherwise. Given the type `Tᵢ` designated by `std::tuple_element<i, E>::type` and the type `Uᵢ` designated by either `Tᵢ&` or `Tᵢ&&`, where `Uᵢ` is an lvalue reference if the initializer is an lvalue and an rvalue reference otherwise, variables are introduced with unique names `rᵢ` as follows:

```cpp
S Uᵢ, rᵢ = initializer ;
```

Each `vᵢ` is the name of an lvalue of type `Tᵢ` that refers to the object bound to `rᵢ`; the referenced type is `Tᵢ`.

Otherwise, all of `E`'s non-static data members shall be direct members of `E` or of the same base class of `E`, well-formed when named as `e.name` in the context of the structured binding, `E` shall not have an anonymous union member, and the number of elements in the identifier-list shall be equal to the number of non-static data members of `E`. Designating the non-static data members of `E` as `m₀, m₁, m₂, ...` (in declaration order), each `vᵢ` is the name of an lvalue that refers to the member `mᵢ` of `e` and whose type is `cv Tᵢ`, where `Tᵢ` is the declared type of that member; the referenced type is `cv Tᵢ`. The lvalue is a bit-field if that member is a bit-field.

[Example 2:
 structs S \{ int x1 : 2; volatile double y1; \};
 S f();
 const auto [ x, y ] = f();

The type of the id-expression x is "const int", the type of the id-expression y is "const volatile double". — end example

9.7 Enumerations [enum]

9.7.1 Enumeration declarations [dcl.enum]

An enumeration is a distinct type (6.8.3) with named constants. Its name becomes an enum-name within its scope.

enum-name:
  identifier

enum-specifier:
  enum-head \{ enumerator-list\opt \}
  enum-head \{ enumerator-list , \}

enum-head:
  enum-key attribute-specifier-seq\opt enum-head-name\opt enum-base\opt

enum-head-name:
  nested-name-specifier\opt identifier

opaque-enum-declaration:
  enum-key attribute-specifier-seq\opt enum-head-name enum-base\opt ;

enum-key:
  enum
  enum class
  enum struct

enum-base:
  : type-specifier-seq

enumerator-list:
  enumerator-definition
  enumerator-list , enumerator-definition

enumerator-definition:
  enumerator
  enumerator = constant-expression

enumerator:
  identifier attribute-specifier-seq\opt

The optional attribute-specifier-seq in the enum-head and the opaque-enum-declaration appertains to the enumeration; the attributes in that attribute-specifier-seq are thereafter considered attributes of the enumeration whenever it is named. A : following "enum nested-name-specifier\opt identifier" within the decl-specifier-seq of a member-declaration is parsed as part of an enum-base.

[Note 1: This resolves a potential ambiguity between the declaration of an enumeration with an enum-base and the declaration of an unnamed bit-field of enumeration type.

[Example 1:]

struct S \{
  enum E : int \{;
  enum E : int \{ // error: redeclaration of enumeration
  \};

  — end example

  — end note

If the enum-head-name of an opaque-enum-declaration contains a nested-name-specifier, the declaration shall be an explicit specialization (13.9.4).

2 The enumeration type declared with an enum-key of only enum is an unscoped enumeration, and its enumerators are unscoped enumerators. The enum-keys enum class and enum struct are semantically equivalent; an enumeration type declared with one of these is a scoped enumeration, and its enumerators are scoped enumerators. The optional enum-head-name shall not be omitted in the declaration of a scoped enumeration.
The \textit{type-specifier-seq} of an \textit{enum-base} shall name an integral type; any \textit{cv}-qualification is ignored. An \textit{opaque-enum-declaration} declaring an unscoped enumeration shall not omit the \textit{enum-base}. The identifiers in an \textit{enumerator-list} are declared as constants, and can appear wherever constants are required. An \textit{enumerator-definition} with \textit{=} gives the associated \textit{enumerator} the value indicated by the \textit{constant-expression}. If the first \textit{enumerator} has no \textit{initializer}, the value of the corresponding constant is zero. An \textit{enumerator-definition} without an \textit{initializer} gives the \textit{enumerator} the value obtained by increasing the value of the previous \textit{enumerator} by one.

[Example 2:
\begin{verbatim}
enum { a, b, c=0 };
enum { d, e, f=e+2 };
\end{verbatim}
\text{defines } a, c, \text{ and } d \text{ to be zero, } b \text{ and } e \text{ to be 1, and } f \text{ to be 3. — end example}]

The optional \textit{attribute-specifier-seq} in an \textit{enumerator} appertains to that \textit{enumerator}.

An \textit{opaque-enum-declaration} is either a redeclaration of an enumeration in the current scope or a declaration of a new enumeration.

[\textit{Note 2:} An enumeration declared by an \textit{opaque-enum-declaration} has a fixed underlying type and is a complete type. The list of enumerators can be provided in a later redeclaration with an \textit{enum-specifier}. — end note]

A scoped enumeration shall not be later redeclared as unscoped or with a different underlying type. An unscoped enumeration shall not be later redeclared as scoped and each redeclaration shall include an \textit{enum-base} specifying the same underlying type as in the original declaration.

If an \textit{enum-head-name} contains a \textit{nested-name-specifier}, it shall not begin with a \textit{decltype-specifier} and the enclosing \textit{enum-specifier} or \textit{opaque-enum-declaration} shall refer to an enumeration that was previously declared directly in the class or namespace to which the \textit{nested-name-specifier} refers, or in an element of the inline namespace set (9.8.2) of that namespace (i.e., neither inherited nor introduced by a \textit{using-declaration}), and the \textit{enum-specifier} or \textit{opaque-enum-declaration} shall appear in a namespace enclosing the previous declaration.

Each enumeration defines a type that is different from all other types. Each enumeration also has an \textit{underlying type}. The underlying type can be explicitly specified using an \textit{enum-base}. For a scoped enumeration type, the underlying type is \textit{int} if it is not explicitly specified. In both of these cases, the underlying type is said to be \textit{fixed}. Following the closing brace of an \textit{enum-specifier}, each \textit{enumerator} has the type of its enumeration. If the underlying type is fixed, the type of each \textit{enumerator} prior to the closing brace is the underlying type and the \textit{constant-expression} in the \textit{enumerator-definition} shall be a converted constant expression of the underlying type (7.7). If the underlying type is not fixed, the type of each \textit{enumerator} prior to the closing brace is determined as follows:

(5.1) — If an \textit{initializer} is specified for an \textit{enumerator}, the \textit{constant-expression} shall be an integral constant expression (7.7). If the expression has unscoped enumeration type, the \textit{enumerator} has the underlying type of that enumeration type, otherwise it has the same type as the expression.

(5.2) — If no \textit{initializer} is specified for the first \textit{enumerator}, its type is an unspecified signed integral type.

(5.3) — Otherwise the type of the \textit{enumerator} is the same as that of the preceding \textit{enumerator} unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value. If no such type exists, the program is ill-formed.

An \textit{enumerator} whose underlying type is fixed is an incomplete type from its point of declaration (6.4.2) to immediately after its \textit{enum-base} (if any), at which point it becomes a complete type. An \textit{enumerator} whose underlying type is not fixed is an incomplete type from its point of declaration to immediately after the closing \} of its \textit{enum-specifier}, at which point it becomes a complete type.

For an \textit{enumerator} whose underlying type is not fixed, the underlying type is an integral type that can represent all the \textit{enumerator} values defined in the enumeration. If no integral type can represent all the \textit{enumerator} values, the enumeration is ill-formed. It is implementation-defined which integral type is used as the underlying type except that the underlying type shall not be larger than \textit{int} unless the value of an \textit{enumerator} cannot fit in an \textit{int} or \textit{unsigned int}. If the \textit{enumerator-list} is empty, the underlying type is as if the enumeration had a single \textit{enumerator} with value 0.

For an \textit{enumerator} whose underlying type is fixed, the values of the enumeration are the values of the underlying type. Otherwise, the values of the enumeration are the values representable by a hypothetical integral type with minimal width \textit{M} such that all enumerators can be represented. The width of the smallest bit-field large enough to hold all the values of the enumeration type is \textit{M}. It is possible to define an
enumeration that has values not defined by any of its enumerators. If the enumeration is empty, the values of the enumeration are as if the enumeration had a single enumerator with value 0.\footnote{This set of values is used to define promotion and conversion semantics for the enumeration type. It does not preclude an expression of enumeration type from having a value that falls outside this range.}

Two enumeration types are *layout-compatible enumerations* if they have the same underlying type.

The value of an enumerator or an object of an unscoped enumeration type is converted to an integer by integral promotion (7.3.7).

[Example 3:]

```c
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue) // ...
```

makes `color` a type describing various colors, and then declares `col` as an object of that type, and `cp` as a pointer to an object of that type. The possible values of an object of type `color` are `red`, `yellow`, `green`, `blue`; these values can be converted to the integral values 0, 1, 20, and 21. Since enumerations are distinct types, objects of type `color` can be assigned only values of type `color`.

```c
color c = 1; // error: type mismatch, no conversion from int to color
int i = yellow; // OK: yellow converted to integral value 1, integral promotion
```

Note that this implicit `enum` to `int` conversion is not provided for a scoped enumeration:

```c
enum class Col { red, yellow, green };
int x = Col::red; // error: no Col to int conversion
Col y = Col::red;
if (y) {} // error: no Col to bool conversion
```

Each `enum-name` and each unscoped `enumerator` is declared in the scope that immediately contains the `enum-specifier`. Each scoped `enumerator` is declared in the scope of the enumeration. An unnamed enumeration that does not have a typedef name for linkage purposes (9.2.4) and that has a first enumerator is denoted, for linkage purposes (6.6), by its underlying type and its first enumerator; such an enumeration is said to have an enumerated type as a name for linkage purposes. These names obey the scope rules defined for all names in 6.4 and 6.5.

[Note 3: Each unnamed enumeration with no enumerators is a distinct type. — end note]

[Example 4:]

```c
enum direction { left='l', right='r' };
void g() {
    direction d; // OK
    d = left; // OK
    d = direction::right; // OK
}
enum class altitude { high='h', low='l' };
void h() {
    altitude a; // OK
    a = high; // error: high not in scope
    a = altitude::low; // OK
}
```

—end example]

[Note 4: An enumerator declared in class scope can be referred to using the class member access operators . (dot) and \texttt{->} (arrow) (7.6.1.5), or with the scope resolution operator :: (6.5.4.2).

[Example 5:]

```c
struct X {
    enum direction { left='l', right='r' };
    int f(int i) { return i==left ? 0 : i==right ? 1 : 2; }
};
```
void g(X* p) {
    direction d; // error: direction not in scope
    int i;
    i = p->f(left); // error: left not in scope
    i = p->f(X::right); // OK
    i = p->f(p->left); // OK
    // ...
}  
—end example] 
—end note] 

9.7.2 The using enum declaration [enum.udecl]  

using enum declaration:  
using elaborated-enum-specifier;  

1 The elaborated-enum-specifier shall not name a dependent type and the type shall have a reachable enum-specifier. 

2 A using-enum-declaration introduces the enumerator names of the named enumeration as if by a using-declaration for each enumerator. 

3 [Note 1: A using-enum-declaration in class scope adds the enumerators of the named enumeration as members to the scope. This means they are accessible for member lookup.  
[Example 1:  
enum class fruit { orange, apple }; 
struct S {  
    using enum fruit; // OK, introduces orange and apple into S 
}; 
void f() {  
    S s;  
    s.orange; // OK, names fruit::orange  
    S::orange; // OK, names fruit::orange 
}  
—end example]  
—end note]  

4 [Note 2: Two using-enum-declarations that introduce two enumerators of the same name conflict.  
[Example 2:  
enum class fruit { orange, apple }; 
enum class color { red, orange }; 
void f() {  
    using enum fruit; // OK  
    using enum color; // error: color::orange and fruit::orange conflict 
}  
—end example]  
—end note] 

9.8 Namespaces [basic.namespace]  

9.8.1 General [basic.namespace.general]  

1 A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units. 

2 [Note 1: A namespace with external linkage is exported if any of its namespace-definitions is exported, or if it contains any export-declarations (10.2). A namespace is never attached to a module, and never has module linkage even if it is not exported. —end note]  
[Example 1:  
export module M;  
namespace N1 {} // N1 is not exported  
export namespace N2 {} // N2 is exported 

§ 9.8.1
namespace N3 { export int n; } // N3 is exported
—end example]

3 The outermost declarative region of a translation unit is a namespace; see 6.4.6.

9.8.2 Namespace definition

9.8.2.1 General

namespace-name:
  identifier
  namespace-alias

namespace-definition:
  named-namescope-definition
  unnamed-name-definition
  nested-namescope-definition

named-namescope-definition:
  inline_opt namespace attribute-specifier-seq_opt identifier { namespace-body }

unnamed-namescope-definition:
  inline_opt namespace attribute-specifier-seq_opt { namespace-body }

nested-namescope-definition:
  namespace enclosing-namescope-specifier :: inline_opt identifier { namespace-body }

enclosing-namescope-specifier:
  identifier
  enclosing-namescope-specifier :: inline_opt identifier

namespace-body:
  declaration-seq_opt

1 Every namespace-definition shall appear at namespace scope (6.4.6).

2 In a named-namescope-definition, the identifier is the name of the namespace. If the identifier, when looked up (6.5.2), refers to a namespace-name (but not a namespace-alias) that was introduced in the namespace in which the named-namescope-definition appears or that was introduced in a member of the inline namespace set of that namespace, the namespace-definition extends the previously-declared namespace. Otherwise, the identifier is introduced as a namespace-name into the declarative region in which the named-namescope-definition appears.

3 Because a namespace-definition contains declarations in its namespace-body and a namespace-definition is itself a declaration, it follows that namespace-declarations can be nested.

[Example 1:

namespace Outer {
  int i;
  namespace Inner {
    void f() { i++; }  // Outer::i
    int i;
    void g() { i++; }  // Inner::i
  }
}
—end example]

4 The enclosing namespaces of a declaration are those namespaces in which the declaration lexically appears, except for a redeclaration of a namespace member outside its original namespace (e.g., a definition as specified in 9.8.2.3). Such a redeclaration has the same enclosing namespaces as the original declaration.

[Example 2:

namespace Q {
  namespace V {
    void f();  // enclosing namespaces are the global namespace, Q, and Q::V
    class C { void m(); 
  }
  void V::f() {
    // enclosing namespaces are the global namespace, Q, and Q::V
    extern void h();  // ... so this declares Q::V::h
  }
}
If the optional initial `inline` keyword appears in a `namespace-definition` for a particular namespace, that namespace is declared to be an `inline namespace`. The `inline` keyword may be used on a `namespace-definition` that extends a namespace only if it was previously used on the `namespace-definition` that initially declared the `namespace-name` for that namespace.

The optional `attribute-specifier-seq` in a `named-name-space-definition` appertains to the namespace being defined or extended.

Members of an inline namespace can be used in most respects as though they were members of the enclosing namespace. Specifically, the inline namespace and its enclosing namespace are both added to the set of associated namespaces used in argument-dependent lookup (6.5.3) whenever one of them is, and a `using-directive` (9.8.4) that names the inline namespace is implicitly inserted into the enclosing namespace as for an unnamed namespace (9.8.2.2). Furthermore, each member of the inline namespace can subsequently be partially specialized (13.7.6), explicitly instantiated (13.9.3), or explicitly specialized (13.9.4) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via explicit qualification (6.5.4.3) will include members of the inline namespace brought in by the `using-directive` even if there are declarations of that name in the enclosing namespace.

These properties are transitive: if a namespace `N` contains an inline namespace `M`, which in turn contains an inline namespace `O`, then the members of `O` can be used as though they were members of `M` or `N`. The `inline namespace set` of `N` is the transitive closure of all inline namespaces in `N`. The `enclosing namespace set` of `O` is the set of namespaces consisting of the innermost non-inline namespace enclosing an inline namespace `O`, together with any intervening inline namespaces.

A `nested-namespace-definition` with an `enclosing-namespace-specifier` `E`, `identifier` `I` and `namespace-body` `B` is equivalent to

```cpp
namespace E { inline_opt namespace I { B } }
```

where the optional `inline` appears if and only if it appears in the `unnamed-namespace-definition` and all occurrences of `unique` in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the translation unit. The optional `attribute-specifier-seq` in the `unnamed-namespace-definition` appertains to `unique`.

```
[Example 1:
namespace { int i; } // unique::i
void f() { i++; } // unique::i++
```
namespace A {
namespace {
  int i;       // A::unique::i
  int j;       // A::unique::j
}
void g() { i++; }       // A::unique::i++
}
using namespace A;
void h() {
  i++;            // error: unique::i or A::unique::i
  A::i++;         // A::unique::i
  j++;            // A::unique::j
}
  —end example]

9.8.2.3 Namespace member definitions
  [namespace.memdef]

1 A declaration in a namespace $N$ (excluding declarations in nested scopes) whose declarator-id is an unqualified-id (9.3.4), whose class-head-name (11.1) or enum-head-name (9.7.1) is an identifier, or whose elaborated-type-specifier is of the form class-key attribute-specifier-seq_opt identifier (9.2.9.4), or that is an opaque-enum-declaration, declares (or redeclares) its unqualified-id or identifier as a member of $N$.

[Note 1: An explicit instantiation (13.9.3) or explicit specialization (13.9.4) of a template does not introduce a name and thus can be declared using an unqualified-id in a member of the enclosing namespace set, if the primary template is declared in an inline namespace. — end note]

[Example 1:

```c
namespace X {
  void f() { /* ... */ }       // OK: introduces X::f()
}
namespace M {
  void g();                    // OK: introduces X::M::g()
} using M::g;
void g();  // error: conflicts with X::M::g()
}

—end example]
```

2 Members of a named namespace can also be defined outside that namespace by explicit qualification (6.5.4.3) of the name being defined, provided that the entity being defined was already declared in the namespace and the definition appears after the point of declaration in a namespace that encloses the declaration’s namespace.

[Example 2:

```c
namespace Q {
  namespace V {
    void f();
  }
  void V::f() { /* ... */ }      // OK
  void V::g() { /* ... */ }      // error: g() is not yet a member of V
} using V::g;
void g();  // error: V doesn’t enclose Q
}

—end example]
```

3 If a friend declaration in a non-local class first declares a class, function, class template or function template the friend is a member of the innermost enclosing namespace. The friend declaration does not by itself make the name visible to unqualified lookup (6.5.2) or qualified lookup (6.5.4).

(99) this implies that the name of the class or function is unqualified.
[Note 2: The name of the friend will be visible in its namespace if a matching declaration is provided at namespace scope (either before or after the class definition granting friendship). — end note]

If a friend function or function template is called, its name may be found by the name lookup that considers functions from namespaces and classes associated with the types of the function arguments (6.5.3). If the name in a friend declaration is neither qualified nor a template-id and the declaration is a function or an elaborated-type-specifier, the lookup to determine whether the entity has been previously declared shall not consider any scopes outside the innermost enclosing namespace.

[Note 3: The other forms of friend declarations cannot declare a new member of the innermost enclosing namespace and thus follow the usual lookup rules. — end note]

[Example 3:

// Assume f and g have not yet been declared.
void h(int);
template <class T> void f2(T);
namespace A {
class X {
friend void f(X); // A::f(X) is a friend
class Y {
friend void g(); // A::g is a friend
friend void h(int); // A::h is a friend
friend void f2<(int); // ::f2<(int) is a friend
};
};

// A::f, A::g and A::h are not visible here
X x;
void g() { f(x); } // definition of A::g
void f(X) { /* ... */ } // definition of A::f
void h(int) { /* ... */ } // definition of A::h

// A::f, A::g and A::h are visible here and known to be friends
}

using A::x;
void h() {
A::f(x);
A::X::f(x); // error: f is not a member of A::X
A::X::Y::g(); // error: g is not a member of A::X::Y
}
—end example]

9.8.3 Namespace alias

A namespace-alias-definition declares an alternate name for a namespace according to the following grammar:

namespace-alias-definition:
  identifier
namespace-alias-definition:
  namespace identifier = qualified-name-specifier;
qualified-name-specifier:
  nested-name-specifier_opt namespace-name

The identifier in a namespace-alias-definition is a synonym for the name of the namespace denoted by the qualified-name-specifier and becomes a namespace-alias.

[Note 1: When looking up a namespace-name in a namespace-alias-definition, only namespace names are considered, see 6.5.7. — end note]

3 In a declarative region, a namespace-alias-definition can be used to redefine a namespace-alias declared in that declarative region to refer only to the namespace to which it already refers.

[Example 1: The following declarations are well-formed:

namespace Company_with_very_long_name { /* ... */ }
namespace CWVLN = Company_with_very_long_name;
namespace CWVLN = Company_with_very_long_name;  // OK: duplicate
namespace CWVLN = CWVLN;

—end example

9.8.4 Using namespace directive

using-directive:
    attribute-specifier-seq_opt using namespace nested-name-specifier_opt namespace-name ;

1 A using-directive shall not appear in class scope, but may appear in namespace scope or in block scope.

[Note 1: When looking up a namespace-name in a using-directive, only namespace names are considered, see 6.5.7. —end note]

The optional attribute-specifier-seq appertains to the using-directive.

2 A using-directive specifies that the names in the nominated namespace can be used in the scope in which the using-directive appears after the using-directive. During unqualified name lookup (6.5.2), the names appear as if they were declared in the nearest enclosing namespace which contains both the using-directive and the nominated namespace.

[Note 2: In this context, “contains” means “contains directly or indirectly”. —end note]

3 A using-directive does not add any members to the declarative region in which it appears.

[Example 1:

namespace A {
    int i;
    namespace B {
        namespace C {
            int i;
        }
        using namespace A::B::C;
        void f1() {
            i = 5;    // OK, C::i visible in B and hides A::i
        }
    }
    namespace D {
        using namespace B;
        using namespace C;
        void f2() {
            i = 5;    // ambiguous, B::C::i or A::i?
        }
    }
    void f3() {
        i = 5;    // uses A::i
    }
    void f4() {
        i = 5;    // error: neither i is visible
    }
}

—end example]

4 For unqualified lookup (6.5.2), the using-directive is transitive: if a scope contains a using-directive that nominates a second namespace that itself contains using-directives, the effect is as if the using-directives from the second namespace also appeared in the first.

[Note 3: For qualified lookup, see 6.5.4.3. —end note]

[Example 2:

namespace M {
    int i;
}

namespace N {
    int i;
}

§ 9.8.4
© ISO/IEC  

using namespace M;
}
void f() {
    using namespace N;
    i = 7;       // error: both M::i and N::i are visible
}

For another example,
namespace A {
    int i;
}
namespace B {
    int i;
    int j;
    namespace C {
        namespace D {
            using namespace A;
            int j;
            int k;
            int a = i; // B::i hides A::i
        }
        using namespace D;
        int k = 89; // no problem yet
        int l = k; // ambiguous: C::k or D::k
        int m = i; // B::i hides A::i
        int n = j; // D::j hides B::j
    }
}
—end example

If a namespace is extended (9.8.2) after a using-directive for that namespace is given, the additional members of the extended namespace and the members of namespaces nominated by using-directives in the extending namespace-definition can be used after the extending namespace-definition.

[Note 4: If name lookup finds a declaration for a name in two different namespaces, and the declarations do not declare the same entity and do not declare functions or function templates, the use of the name is ill-formed (6.5). In particular, the name of a variable, function or enumerator does not hide the name of a class or enumeration declared in a different namespace. For example,
namespace A {
    class X { }
    extern "C" int g();
    extern "C++" int h();
}
namespace B {
    void X(int);
    extern "C" int g();
    extern "C++" int h(int);
}
using namespace A;
using namespace B;
void f() {
    X(1);       // error: name X found in two namespaces
    g();       // OK: name g refers to the same entity
    h();       // OK: overload resolution selects A::h
}
—end note]  

During overload resolution, all functions from the transitive search are considered for argument matching. The set of declarations found by the transitive search is unordered.

[Note 5: In particular, the order in which namespaces were considered and the relationships among the namespaces implied by the using-directives do not cause preference to be given to any of the declarations found by the search.  
—end note]

§ 9.8.4
An ambiguity exists if the best match finds two functions with the same signature, even if one is in a namespace reachable through using-directives in the namespace of the other.\footnote{During name lookup in a class hierarchy, some ambiguities can be resolved by considering whether one member hides the other along some paths (11.8). There is no such disambiguation when considering the set of names found as a result of following using-directives.}

[Example 3:]

```cpp
namespace D {
    int d1;
    void f(char);
}
using namespace D;
int d1;  // OK: no conflict with D::d1

namespace E {
    int e;
    void f(int);
}

namespace D {  // namespace extension
    int d2;
    using namespace E;
    void f(int);
}

void f() {
    d1++;       // error: ambiguous ::d1 or D::d1?
    ::d1++;
    D::d1++;  // OK
    d2++;       // OK: D::d2
    e++;        // OK: E::e
    f(1);       // error: ambiguous: D::f(int) or E::f(int)?
    f('a');    // OK: D::f(char)
}

—end example]

9.9 The using declaration

```cpp
using-declaration:
    using using-declarator-list ;

using-declarator-list:
    using-declarator ...\texttt{opt}
    using-declarator-list , using-declarator ...\texttt{opt}

using-declarator:
    \texttt{typename\texttt{opt}} nested-name-specifier unqualified-id
```

Each using-declarator in a using-declaration\footnote{A using-declaration with more than one using-declarator is equivalent to a corresponding sequence of using-declarations with one using-declarator each.} introduces a set of declarations into the declarative region in which the using-declaration appears. The set of declarations introduced by the using-declarator is found by performing qualified name lookup (6.5.4, 11.8) for the name in the using-declarator, excluding functions that are hidden as described below. If the using-declarator does not name a constructor, the unqualified-id is declared in the declarative region in which the using-declaration appears as a synonym for each declaration introduced by the using-declarator.

[Note 1: Only the specified name is so declared; specifying an enumeration name in a using-declaration does not declare its enumerators in the using-declaration's declarative region. —end note]

If the using-declarator names a constructor, it declares that the class inherits the set of constructor declarations introduced by the using-declarator from the nominated base class.

\footnote{Every using-declaration is a declaration and a member-declaration and can therefore be used in a class definition.}

[Example 1:]

```cpp
}
struct B {
  void f(char);
  void g(char);
  enum E { e };
  union { int x; };
};

struct D : B {
  using B::f;
  void f(int) { f('c'); } // calls B::f(char)
  void g(int) { g('c'); } // recursively calls D::g(int)
};

—end example]

3 In a using-declaration used as a member-declaration, each using-declarator shall either name an enumerator or have a nested-name-specifier naming a base class of the class being defined.

[Example 2:
enum class button { up, down };
struct S {
  using button::up;
  button b = up;
  // OK
};
—end example]

If a using-declarator names a constructor, its nested-name-specifier shall name a direct base class of the class being defined.

[Example 3:
template <typename... bases>
struct X : bases... {
  using bases::g...;
};
X<B, D> x; // OK: B::g and D::g introduced
—end example]

[Example 4:
class C {
  int g();
};

class D2 : public B {
  using B::f; // OK: B is a base of D2
  using B::e; // OK: e is an enumerator of base B
  using B::x; // OK: x is a union member of base B
  using C::g; // error: C isn't a base of D2
};
—end example]

4 [Note 2: Since destructors do not have names, a using-declaration cannot refer to a destructor for a base class. Since specializations of member templates for conversion functions are not found by name lookup, they are not considered when a using-declaration specifies a conversion function (13.7.3). — end note]

If a constructor or assignment operator brought from a base class into a derived class has the signature of a copy/move constructor or assignment operator for the derived class (11.4.5.3, 11.4.6), the using-declaration does not by itself suppress the implicit declaration of the derived class member; the member from the base class is hidden or overridden by the implicitly-declared copy/move constructor or assignment operator of the derived class, as described below.

5 A using-declaration shall not name a template-id.

[Example 5:
struct A {
  template <class T> void f(T);
]
template <class T> struct X { }
;
struct B : A {
    using A::f<double>; // error
    using A::X<int>;   // error
};
—end example]

6 A using-declaration shall not name a namespace.

7 A using-declaration that names a class member other than an enumerator shall be a member-declaration.
[Example 6:
struct X {
    int i;
    static int s;
};

void f() {
    using X::i;          // error: X::i is a class member and this is not a member declaration.
    using X::s;          // error: X::s is a class member and this is not a member declaration.
}
—end example]

8 Members declared by a using-declaration can be referred to by explicit qualification just like other member names (6.5.4.3).
[Example 7:
void f();

namespace A {
    void g();
}
namespace X {
    using ::f;           // global f
    using A::g;          // A's g
}

void h()
{
    X::f();              // calls ::f
    X::g();              // calls A::g
}
—end example]

9 A using-declaration is a declaration and can therefore be used repeatedly where (and only where) multiple declarations are allowed.
[Example 8:
namespace A {
    int i;
}

namespace A1 {
    using A::i, A::i;    // OK: double declaration
}

struct B {
    int i;
};

struct X : B {
    using B::i, B::i;   // error: double member declaration
};
[Note 3: For a using-declaration whose nested-name-specifier names a namespace, members added to the namespace after the using-declaration are not in the set of introduced declarations, so they are not considered when a use of the name is made. Thus, additional overloads added after the using-declaration are ignored, but default function arguments (9.3.4.7), default template arguments (13.2), and template specializations (13.7.6, 13.9.4) are considered.
—end note]

Example 9:

```cpp	namespace A {
  void f(int);
}

using A::f;  // f is a synonym for A::f; that is, for A::f(int).
namespace A {
  void f(char);
}

void foo() {
  f('a');  // calls f(int), even though f(char) exists.
}

void bar() {
  using A::f;  // f is a synonym for A::f; that is, for A::f(int) and A::f(char).
  f('a');  // calls f(char)
}

—end example]

[Note 4: Partial specializations of class templates are found by looking up the primary class template and then considering all partial specializations of that template. If a using-declaration names a class template, partial specializations introduced after the using-declaration are effectively visible because the primary template is visible (13.7.6). —end note]

Since a using-declaration is a declaration, the restrictions on declarations of the same name in the same declarative region (6.4) also apply to using-declarations.

Example 10:

```cpp	namespace A {
  int x;
}

namespace B {
  int i;
  struct g { };  // OK: hides struct g
  struct x { }
  void f(int);  // OK: hides struct g
  void f(double);
  void g(char);  // OK: hides struct g
}

void func() {
  int i;
  using B::i;  // error: i declared twice
  void f(char);
  using B::f;  // OK: each f is a function
  f(3.5);  // calls B::f(double)
  using B::g;
  g('a');  // calls B::g(char)
  struct g g1;  // g1 has class type B::g
  using B::x;
  using A::x;  // OK: hides struct B::x
  x = 99;  // assigns to A::x
  struct x x1;  // x1 has class type B::x
}

—end example]
13 If a function declaration in namespace scope or block scope has the same name and the same parameter-type-list (9.3.4.6) as a function introduced by a using-declaration, and the declarations do not declare the same function, the program is ill-formed. If a function template declaration in namespace scope has the same name, parameter-type-list, trailing requires-clause (if any), return type, and template-head, as a function template introduced by a using-declaration, the program is ill-formed.

[Note 5: Two using-declarations can introduce functions with the same name and the same parameter-type-list. If, for a call to an unqualified function name, function overload resolution selects the functions introduced by such using-declarations, the function call is ill-formed.]

[Example 11:]

```cpp
namespace B {
    void f(int);
    void f(double);
}
namespace C {
    void f(int);
    void f(double);
    void f(char);
}

void h() {
    using B::f; // B::f(int) and B::f(double)
    using C::f; // C::f(int), C::f(double), and C::f(char)
    f('h'); // calls C::f(char)
    f(1); // error: ambiguous: B::f(int) or C::f(int)?
    void f(int); // error: f(int) conflicts with C::f(int) and B::f(int)
}
```

—end example]
—end note]

14 When a using-declarator brings declarations from a base class into a derived class, member functions and member function templates in the derived class override and/or hide member functions and member function templates with the same name, parameter-type-list (9.3.4.6), trailing requires-clause (if any), cv-qualification, and ref-qualifier (if any), in a base class (rather than conflicting). Such hidden or overridden declarations are excluded from the set of declarations introduced by the using-declarator.

[Example 12:]

```cpp
struct B {
    virtual void f(int);
    virtual void f(char);
    void g(int);
    void h(int);
};

struct D : B {
    using B::f;
    void f(int); // OK: D::f(int) overrides B::f(int);
    using B::g;
    void g(char); // OK
    using B::h;
    void h(int); // OK: D::h(int) hides B::h(int)
};

void k(D* p) {
    p->f(1); // calls D::f(int)
    p->f('a'); // calls B::f(char)
    p->g(1); // calls B::g(int)
    p->g('a'); // calls D::g(char)
}
```

§ 9.9
struct B1 {
    B1(int);
};

struct B2 {
    B2(int);
};

struct D1 : B1, B2 {
    using B1::B1;
    using B2::B2;
};
D1 d1(0); // error: ambiguous

struct D2 : B1, B2 {
    using B1::B1;
    using B2::B2;
    D2(int); // OK: D2::D2(int) hides B1::B1(int) and B2::B2(int)
};
D2 d2(0); // calls D2::D2(int)

— end example

[Note 6: For the purpose of forming a set of candidates during overload resolution, the functions that are introduced by a using-declaration into a derived class are treated as though they were members of the derived class (11.8). In particular, the implicit object parameter is treated as if it were a reference to the derived class rather than to the base class (12.4.2). This has no effect on the type of the function, and in all other respects the function remains a member of the base class. — end note]

Constructors that are introduced by a using-declaration are treated as though they were constructors of the derived class when looking up the constructors of the derived class (6.5.4.2) or forming a set of overload candidates (12.4.2.4, 12.4.2.5, 12.4.2.8).

[Note 7: If such a constructor is selected to perform the initialization of an object of class type, all subobjects other than the base class from which the constructor originated are implicitly initialized (11.10.4). A constructor of a derived class is sometimes preferred to a constructor of a base class if they would otherwise be ambiguous (12.4.4). — end note]

In a using-declarator that does not name a constructor, all members of the set of introduced declarations shall be accessible. In a using-declarator that names a constructor, no access check is performed. In particular, if a derived class uses a using-declarator to access a member of a base class, the member name shall be accessible. If the name is that of an overloaded member function, then all functions named shall be accessible. The base class members mentioned by a using-declarator shall be visible in the scope of at least one of the direct base classes of the class where the using-declarator is specified.

[Note 8: Because a using-declarator designates a base class member (and not a member subobject or a member function of a base class subobject), a using-declarator cannot be used to resolve inherited member ambiguities.

[Example 13:

struct A { int x(); };
struct B : A { };
struct C : A {
    using A::x;
    int x(int);
};

struct D : B, C {
    using C::x;
    int x(double);
};
int f(D* d) {
    return d->x(); // error: overload resolution selects A::x, but A is an ambiguous base class
}
— end example
— end note]
A synonym created by a using-declaration has the usual accessibility for a member-declaration. A using-declarator that names a constructor does not create a synonym; instead, the additional constructors are accessible if they would be accessible when used to construct an object of the corresponding base class, and the accessibility of the using-declaration is ignored.

[Example 14:

```cpp
class A {
private:
    void f(char);
public:
    void f(int);
protected:
    void g();
};

class B : public A {
    using A::f; // error: A::f(char) is inaccessible
    public:
    using A::g; // B::g is a public synonym for A::g
};

-end example]

If a using-declarator uses the keyword typename and specifies a dependent name (13.8.3), the name introduced by the using-declaration is treated as a typedef-name (9.2.4).

9.10 The asm declaration [dcl.asm]

An asm declaration has the form

```cpp
asm-declaration:
    attribute-specifier-seq_opt asm ( string-literal ) ;
```

The asm declaration is conditionally-supported; its meaning is implementation-defined. The optional attribute-specifier-seq in an asm-declaration appertains to the asm declaration.

[Note 1: Typically it is used to pass information through the implementation to an assembler. — end note]

9.11 Linkage specifications [dcl.link]

All function types, function names with external linkage, and variable names with external linkage have a language linkage.

[Note 1: Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here. For example, a particular language linkage might be associated with a particular form of representing names of objects and functions with external linkage, or with a particular calling convention, etc. — end note]

The default language linkage of all function types, function names, and variable names is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.

Linkage (6.6) between C++ and non-C++ code fragments can be achieved using a linkage-specification:

```cpp
linkage-specification:
    extern string-literal { declaration-seq_opt }
    extern string-literal declaration
```

The string-literal indicates the required language linkage. This document specifies the semantics for the string-literals "C" and "C++". Use of a string-literal other than "C" or "C++" is conditionally-supported, with implementation-defined semantics.

[Note 2: Therefore, a linkage-specification with a string-literal that is unknown to the implementation requires a diagnostic. — end note]

[Note 3: It is recommended that the spelling of the string-literal be taken from the document defining that language. For example, Ada (not ADA) and Fortran or FORTRAN, depending on the vintage. — end note]

Every implementation shall provide for linkage to functions written in the C programming language, "C", and linkage to C++ functions, "C++".

[Example 1:}
A module-import-declaration shall not be directly contained in a linkage-specification. A module-import-declaration appearing in a linkage specification with other than C++ language linkage is conditionally-supported with implementation-defined semantics.

5 Linkage specifications nest. When linkage specifications nest, the innermost one determines the language linkage. A linkage specification does not establish a scope. A linkage-specification shall occur only in namespace scope (6.4). In a linkage-specification, the specified language linkage applies to the function types of all function declarators, function names with external linkage, and variable names with external linkage declared within the linkage-specification.

Example 2:

```cpp
extern "C" // the name f1 and its function type have C language linkage;
void f1(void(*pf)(int)); // pf is a pointer to a C function

FUNC f2; // the name f2 has C++ language linkage and the
// function's type has C language linkage

extern "C" FUNC f3; // the name of function f3 and the function's type have C language linkage

void (*pf2)(FUNC*); // the name of the variable pf2 has C++ linkage and the type
// of pf2 is “pointer to C++ function that takes one parameter of type
// pointer to C function”

extern "C" { // the name of the function f4 has internal linkage (not C language linkage)
    static void f4(); // and the function's type has C language linkage.
}

extern "C" void f5() { // OK: Name linkage (internal) and function type linkage (C language linkage)
    extern void f4(); // obtained from previous declaration.
}

extern void f4(); // OK: Name linkage (internal) and function type linkage (C language linkage)
// obtained from previous declaration.

void f6() { // OK: Name linkage (internal) and function type linkage (C language linkage)
    extern void f4(); // obtained from previous declaration.
}
```

—end example]

A C language linkage is ignored in determining the language linkage of the names of class members and the function type of class member functions.

Example 3:

```cpp
extern "C" typedef void FUNC_c();

class C {
    void mf1(FUNC_c*); // the name of the function mf1 and the member function's type have
    // C++ language linkage; the parameter has type “pointer to C function”

    FUNC_c mf2; // the name of the function mf2 and the member function's type have
    // C++ language linkage

    static FUNC_c* q; // the name of the data member q has C++ language linkage and
    // the data member's type is “pointer to C function”
};
```
extern "C" {
  class X {
    void mf();          // the name of the function mf and the member function's type have
                      // C++ language linkage
    void mf2(void(*)()); // the name of the function mf2 has C++ language linkage;
                      // the parameter has type "pointer to C function"
  }
};
}  // end example

If two declarations declare functions with the same name and parameter-type-list (9.3.4.6) to be members of the same namespace or declare objects with the same name to be members of the same namespace and the declarations give the names different language linkages, the program is ill-formed; no diagnostic is required if the declarations appear in different translation units. Except for functions with C++ linkage, a function declaration without a linkage specification shall not precede the first linkage specification for that function. A function can be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

At most one function with a particular name can have C language linkage. Two declarations for a function with C language linkage with the same function name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same function. Two declarations for a variable with C language linkage with the same name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same variable. An entity with C language linkage shall not be declared with the same name as a variable in global scope, unless both declarations denote the same entity; no diagnostic is required if the declarations appear in different translation units. A variable with C language linkage shall not be declared with the same name as a function with C language linkage (ignoring the namespace names that qualify the respective names); no diagnostic is required if the declarations appear in different translation units.

[Note 4: Only one definition for an entity with a given name with C language linkage can appear in the program (see 6.3); this implies that such an entity must not be defined in more than one namespace scope. —end note]

[Example 4:
	n int x;

namespace A {
  extern "C" int f();
  extern "C" int g() { return 1; }
  extern "C" int h();
  extern "C" int x(); // error: same name as global-space object x
}

namespace B {
  extern "C" int f(); // A::f and B::f refer to the same function
  extern "C" int g() { return 1; } // error: the function g with C language linkage has two definitions
}

int A::f() { return 98; } // definition for the function f with C language linkage
extern "C" int h() { return 97; } // A::h and ::h refer to the same function

—end example]

A declaration directly contained in a linkage-specification is treated as if it contains the extern specifier (9.2.2) for the purpose of determining the linkage of the declared name and whether it is a definition. Such a declaration shall not specify a storage class.

[Example 5:

extern "C" double f(); // error
static double f(); // declaration
extern "C" int i; // declaration
extern "C" {
  int i; // definition
}

extern "C" static void g(); // error

—end example]
[Note 5: Because the language linkage is part of a function type, when indrecting through a pointer to C function, the function to which the resulting lvalue refers is considered a C function. — end note]

Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved.

9.12 Attributes
9.12.1 Attribute syntax and semantics

Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

```
attribute-specifier-seq:
  attribute-specifier-seq_opt attribute-specifier

attribute-specifier:
  [ [ attribute-using-prefix_opt attribute-list ] ]
  alignment-specifier

alignment-specifier:
  alignas ( type-id ..._opt )
  alignas ( constant-expression ..._opt )

attribute-using-prefix:
  using attribute-namespaced :

attribute-list:
  attributeopt
  attribute-list , attributeopt
  attribute ... attribute-list , attribute ...

attribute:
  attribute-token attribute-argument-clause_opt

attribute-token:
  identifier
  attribute-scoped-token

attribute-scoped-token:
  attribute-namespaced : identifier

attribute-namespace:
  identifier

attribute-argument-clause:
  ( balanced-token-seq_opt )

balanced-token-seq:
  balanced-token
  balanced-token-seq balanced-token

balanced-token:
  ( balanced-token-seq_opt )
  [ balanced-token-seq_opt ]
  { balanced-token-seq_opt }

any token other than a parenthesis, a bracket, or a brace
```

If an attribute-specifier contains an attribute-using-prefix, the attribute-list following that attribute-using-prefix shall not contain an attribute-scoped-token and every attribute-token in that attribute-list is treated as if its identifier were prefixed with N::, where N is the attribute-namespaced specified in the attribute-using-prefix.

[Note 1: This rule imposes no constraints on how an attribute-using-prefix affects the tokens in an attribute-argument-clause. — end note]

[Example 1:]

```
[[using CC: opt(1), debug]] // same as [[CC::opt(1), CC::debug]]
void f() {}
[[using CC: opt(1)]] [[CC::debug]] // same as [[CC::opt(1)]] [[CC::debug]]
void g() {}
[[using CC: CC::opt(1)]] // error: cannot combine using and scoped attribute token
void h() {}
```
Note 2: For each individual attribute, the form of the balanced-token-seq will be specified. — end note

In an attribute-list, an ellipsis may appear only if that attribute's specification permits it. An attribute followed by an ellipsis is a pack expansion (13.7.4). An attribute-specifier that contains no attributes has no effect. The order in which the attribute-tokens appear in an attribute-list is not significant. If a keyword (5.11) or an alternative token (5.5) that satisfies the syntactic requirements of an identifier (5.10) is contained in an attribute-token, it is considered an identifier. No name lookup (6.5) is performed on any of the identifiers contained in an attribute-token. The attribute-token determines additional requirements on the attribute-argument-clause (if any).

Each attribute-specifier-seq is said to appertain to some entity or statement, identified by the syntactic context where it appears (Clause 8, Clause 9, 9.3). If an attribute-specifier-seq that appertains to some entity or statement contains an attribute or alignment-specifier that is not allowed to apply to that entity or statement, the program is ill-formed. If an attribute-specifier-seq appertains to a friend declaration (11.9.4), that declaration shall be a definition.

Note 3: An attribute-specifier-seq cannot appertain to an explicit instantiation (13.9.3). — end note

For an attribute-token (including an attribute-scoped-token) not specified in this document, the behavior is implementation-defined. Any attribute-token that is not recognized by the implementation is ignored. An attribute-token is reserved for future standardization if

(6.1) it is not an attribute-scoped-token and is not specified in this document, or

(6.2) it is an attribute-scoped-token and its attribute-namespace is std followed by zero or more digits.

Each implementation should choose a distinctive name for the attribute-namespace in an attribute-scoped-token.

Two consecutive left square bracket tokens shall appear only when introducing an attribute-specifier or within the balanced-token-seq of an attribute-argument-clause.

Note 4: If two consecutive left square brackets appear where an attribute-specifier is not allowed, the program is ill-formed even if the brackets match an alternative grammar production. — end note

Example 2:

```c
int p[10];
void f() {
    int x = 42, y[5];
    int(p[x] { return x; }()); // error: invalid attribute on a nested declarator-id and
    // not a function-style cast of an element of p.
    y[] { return 2; }() = 2; // error even though attributes are not allowed in this context.
    int i ([vendor::attr([])]); // well-formed implementation-defined attribute.
}
```

— end example

9.12.2 Alignment specifier

An alignment-specifier may be applied to a variable or to a class data member, but it shall not be applied to a bit-field, a function parameter, or an exception-declaration (14.4). An alignment-specifier may also be applied to the declaration of a class (in an elaborated-type-specifier (9.2.9.4) or class-head (Clause 11), respectively). An alignment-specifier with an ellipsis is a pack expansion (13.7.4).

When the alignment-specifier is of the form alignas( constant-expression ):

(2.1) the constant-expression shall be an integral constant expression

(2.2) if the constant expression does not evaluate to an alignment value (6.7.6), or evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed.

An alignment-specifier of the form alignas( type-id ) has the same effect as alignas(alignof( type-id )) (7.6.2.6).

The alignment requirement of an entity is the strictest nonzero alignment specified by its alignment-specifiers, if any; otherwise, the alignment-specifiers have no effect.

The combined effect of all alignment-specifiers in a declaration shall not specify an alignment that is less strict than the alignment that would be required for the entity being declared if all alignment-specifiers appertaining to that entity were omitted.
If the defining declaration of an entity has an alignment-specifier, any non-defining declaration of that entity shall either specify equivalent alignment or have no alignment-specifier. Conversely, if any declaration of an entity has an alignment-specifier, every defining declaration of that entity shall specify an equivalent alignment. No diagnostic is required if declarations of an entity have different alignment-specifiers in different translation units.

Example 2:

```c
// Translation unit #1:
struct S { int x; } s, *p = &s;

// Translation unit #2:
struct alignas(16) S; // ill-formed, no diagnostic required: definition of S lacks alignment
extern S* p;
```

Example 3: An aligned buffer with an alignment requirement of A and holding N elements of type T can be declared as:

```c
alignas(T) alignas(A) T buffer[N];
```

Specifying `alignas(T)` ensures that the final requested alignment will not be weaker than `alignof(T)`, and therefore the program will not be ill-formed.

Example 4:

```c
alignas(double) void f(); // error: alignment applied to function
alignas(double) unsigned char c[sizeof(double)]; // array of characters, suitably aligned for a double
extern unsigned char c[sizeof(double)]; // no alignas necessary
alignas(float)
    extern unsigned char c[sizeof(double)]; // error: different alignment in declaration
```

### 9.12.3 Carries dependency attribute

The attribute-token `carries_dependency` specifies dependency propagation into and out of functions. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to the declarator-id of a parameter-declaration in a function declaration or lambda, in which case it specifies that the initialization of the parameter carries a dependency to (6.9.2) each lvalue-to-rvalue conversion (7.3.2) of that object. The attribute may also be applied to the declarator-id of a function declaration, in which case it specifies that the return value, if any, carries a dependency to the evaluation of the function call expression.

The first declaration of a function shall specify the `carries_dependency` attribute for its declarator-id if any declaration of the function specifies the `carries_dependency` attribute. Furthermore, the first declaration of a function shall specify the `carries_dependency` attribute for a parameter if any declaration of that function specifies the `carries_dependency` attribute for that parameter. If a function or one of its parameters is declared with the `carries_dependency` attribute in its first declaration in one translation unit and the same function or one of its parameters is declared without the `carries_dependency` attribute in its first declaration in another translation unit, the program is ill-formed, no diagnostic required.

[Note 1: The `carries_dependency` attribute does not change the meaning of the program, but might result in generation of more efficient code. — end note]

Example 1:

```c
/* Translation unit A. */

struct foo { int* a; int* b; };
std::atomic<struct foo*> foo_head[10];
int foo_array[10][10];
```
[[carries_dependency]] struct foo* f(int i) {
    return foo_head[i].load(memory_order::consume);
}

int g(int* x, int* y [[carries_dependency]]) {
    return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */

[[carries_dependency]] struct foo* f(int i);
int g(int* x, int* y [[carries_dependency]]);
int c = 3;
void h(int i) {
    struct foo* p;
    p = f(i);
    do_something_with(g(&c, p->a));
    do_something_with(g(p->a, &c));
}

The carries_dependency attribute on function f means that the return value carries a dependency out of f, so that the implementation need not constrain ordering upon return from f. Implementations of f and its caller may choose to preserve dependencies instead of emitting hardware memory ordering instructions (a.k.a. fences). Function g’s second parameter has a carries_dependency attribute, but its first parameter does not. Therefore, function h’s first call to g carries a dependency into g, but its second call does not. The implementation might need to insert a fence prior to the second call to g. —end example]

9.12.4 Deprecated attribute [[dcl.attr.deprecated]]
1 The attribute-token deprecated can be used to mark names and entities whose use is still allowed, but is discouraged for some reason.

[Note 1: In particular, deprecated is appropriate for names and entities that are deemed obsolescent or unsafe. —end note]

It shall appear at most once in each attribute-list. An attribute-argument-clause may be present and, if present, it shall have the form:

( string-literal )

[Note 2: The string-literal in the attribute-argument-clause can be used to explain the rationale for deprecation and/or to suggest a replacing entity. —end note]

2 The attribute may be applied to the declaration of a class, a typedef-name, a variable, a non-static data member, a function, a namespace, an enumeration, an enumerator, or a template specialization.

3 A name or entity declared without the deprecated attribute can later be redeclared with the attribute and vice-versa.

[Note 3: Thus, an entity initially declared without the attribute can be marked as deprecated by a subsequent redeclaration. However, after an entity is marked as deprecated, later redeclarations do not un-deprecate the entity. —end note]

Redeclarations using different forms of the attribute (with or without the attribute-argument-clause or with different attribute-argument-clauses) are allowed.

4 Recommended practice: Implementations should use the deprecated attribute to produce a diagnostic message in case the program refers to a name or entity other than to declare it, after a declaration that specifies the attribute. The diagnostic message should include the text provided within the attribute-argument-clause of any deprecated attribute applied to the name or entity.

9.12.5 Fallthrough attribute [[dcl.attr.fallthrough]]
1 The attribute-token fallthrough may be applied to a null statement (8.3); such a statement is a fallthrough statement. The attribute-token fallthrough shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. A fallthrough statement may only appear within an enclosing switch statement (8.5.3). The next statement that would be executed after a fallthrough statement shall be a labeled
statement whose label is a case label or default label for the same switch statement and, if the fallthrough statement is contained in an iteration statement, the next statement shall be part of the same execution of the substatement of the innermost enclosing iteration statement. The program is ill-formed if there is no such statement.

2 **Recommended practice:** The use of a fallthrough statement should suppress a warning that an implementation might otherwise issue for a case or default label that is reachable from another case or default label along some path of execution. Implementations should issue a warning if a fallthrough statement is not dynamically reachable.

3 **Example 1:**

```c
void f(int n) {
    void g(), h(), i();
    switch (n) {
    case 1:
    case 2:
        g();
        [[fallthrough]];  // warning on fallthrough discouraged
    case 3:
        do {
            [[fallthrough]];  // error: next statement is not part of the same substatement execution
        } while (false);
    case 6:
        do {
            [[fallthrough]];  // error: next statement is not part of the same substatement execution
        } while (n--);
    case 7:
        while (false) {
            [[fallthrough]];  // error: next statement is not part of the same substatement execution
        }
    case 5:
        h();
    case 4:
        // implementation may warn on fallthrough
        i();
        [[fallthrough]];  // error
    }
}
```

—end example—

### 9.12.6 Likelihood attributes

The attribute-tokens **likely** and **unlikely** may be applied to labels or statements. The attribute-tokens **likely** and **unlikely** shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute-token **likely** shall not appear in an attribute-specifier-seq that contains the attribute-token **unlikely**.

2 **Recommended practice:** The use of the **likely** attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more likely than any alternative path of execution that does not include such an attribute on a statement or label. The use of the **unlikely** attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more unlikely than any alternative path of execution that does not include such an attribute on a statement or label. A path of execution includes a label if and only if it contains a jump to that label.

[Note 1: Excessive usage of either of these attributes is liable to result in performance degradation. — end note]

3 **Example 1:**

```c
void g(int);
int f(int n) {
    if (n > 5) [[unlikely]] {
        // n > 5 is considered to be arbitrarily unlikely
        g(0);
        return n * 2 + 1;
    }
}
```
switch (n) {
    case 1:
        g(1);
        [[fallthrough]];

        [[likely]] case 2:      // n == 2 is considered to be arbitrarily more
            g(2);                // likely than any other value of n
        break;
    }
    return 3;
}
— end example]

9.12.7 Maybe unused attribute [dcl.attr.unused]
1 The attribute-token maybe_unused indicates that a name or entity is possibly intentionally unused. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present.
2 The attribute may be applied to the declaration of a class, a typedef-name, a variable (including a structured binding declaration), a non-static data member, a function, an enumeration, or an enumerator.
3 A name or entity declared without the maybe_unused attribute can later be redeclared with the attribute and vice versa. An entity is considered marked after the first declaration that marks it.
4 Recommended practice: For an entity marked maybe_unused, implementations should not emit a warning that the entity or its structured bindings (if any) are used or unused. For a structured binding declaration not marked maybe_unused, implementations should not emit such a warning unless all of its structured bindings are unused.
5 [Example 1]:
   [[maybe_unused]] void f([[maybe_unused]] bool thing1,
                               [[maybe_unused]] bool thing2) {
       [[maybe_unused]] bool b = thing1 && thing2;
       assert(b);
   }
Implementations should not warn that b is unused, whether or not NDEBUG is defined. — end example]

9.12.8 Nodiscard attribute [dcl.attr.nodiscard]
1 The attribute-token nodiscard may be applied to the declarator-id in a function declaration or to the declaration of a class or enumeration. It shall appear at most once in each attribute-list. An attribute-argument-clause may be present and, if present, shall have the form:
   (string-literal)
2 A name or entity declared without the nodiscard attribute can later be redeclared with the attribute and vice-versa.
   [Note 1: Thus, an entity initially declared without the attribute can be marked as nodiscard by a subsequent redeclaration. However, after an entity is marked as nodiscard, later redeclarations do not remove the nodiscard from the entity. — end note]
Redeclarations using different forms of the attribute (with or without the attribute-argument-clause or with different attribute-argument-clauses) are allowed.
3 A nodiscard type is a (possibly cv-qualified) class or enumeration type marked nodiscard in a reachable declaration. A nodiscard call is either
   (3.1) — a function call expression (7.6.1.3) that calls a function declared nodiscard in a reachable declaration or whose return type is a nodiscard type, or
   (3.2) — an explicit type conversion (7.6.1.4, 7.6.1.9, 7.6.3) that constructs an object through a constructor declared nodiscard in a reachable declaration, or that initializes an object of a nodiscard type.
4 Recommended practice: Appearance of a nodiscard call as a potentially-evaluated discarded-value expression (7.2) is discouraged unless explicitly cast to void. Implementations should issue a warning in such cases.
   [Note 2: This is typically because discarding the return value of a nodiscard call has surprising consequences. — end note]

§ 9.12.8 245
The string-literal in a \texttt{nodiscard} attribute-argument-clause should be used in the message of the warning as the rationale for why the result should not be discarded.

[Example 1:]

```cpp
struct [[nodiscard]] my_scopeguard { /* ... */};
struct my_unique {
    my_unique() = default; // does not acquire resource
    [[nodiscard]] my_unique(int fd) { /* ... */ } // acquires resource
    my_unique() noexcept { /* ... */ } // releases resource, if any
};
struct [[nodiscard]] error_info { /* ... */};
error_info enable_missile_safety_mode();
void launch_missiles();
void test_missiles() {
    my_scopeguard(); // warning encouraged
    (void)my_scopeguard(), // warning not encouraged, cast to void
    launch_missiles(); // comma operator, statement continues
    my_unique(42); // warning encouraged
    my_unique(); // warning not encouraged
    enable_missile_safety_mode(); // warning encouraged
    launch_missiles();
}
error_info &foo();
void f() { foo(); } // warning not encouraged: not a nodiscard call, because neither
// the (reference) return type nor the function is declared nodiscard

— end example]

9.12.9 Noreturn attribute

The attribute-token \texttt{noreturn} specifies that a function does not return. It shall appear at most once in each \texttt{attribute-list} and no \texttt{attribute-argument-clause} shall be present. The attribute may be applied to the \texttt{declarator-id} in a function declaration. The first declaration of a function shall specify the \texttt{noreturn} attribute if any declaration of that function specifies the \texttt{noreturn} attribute. If a function is declared with the \texttt{noreturn} attribute in one translation unit and the same function is declared without the \texttt{noreturn} attribute in another translation unit, the program is ill-formed, no diagnostic required.

If a function \texttt{f} is called where \texttt{f} was previously declared with the \texttt{noreturn} attribute and \texttt{f} eventually returns, the behavior is undefined.

[Note 1: The function may terminate by throwing an exception. — end note]

Recommended practice: Implementations should issue a warning if a function marked [[\texttt{noreturn}]] might return.

[Example 1:]

```cpp
[[ noreturn ]] void f() {
    throw "error"; // OK
}
```

```cpp
[[ noreturn ]] void q(int i) {
    // behavior is undefined if called with an argument <= 0
    if (i > 0)
        throw "positive";
}

— end example]

9.12.10 No unique address attribute

The attribute-token \texttt{no_unique_address} specifies that a non-static data member is a potentially-overlapping subobject (6.7.2). It shall appear at most once in each \texttt{attribute-list} and no \texttt{attribute-argument-clause} shall be present. The attribute may appertain to a non-static data member other than a bit-field.

[Note 1: The non-static data member can share the address of another non-static data member or that of a base class, and any padding that would normally be inserted at the end of the object can be reused as storage for other members. — end note]

[Example 1:]

```cpp
§ 9.12.10 246
```
template<typename Key, typename Value,
    typename Hash, typename Pred, typename Allocator>
class hash_map {
    [[no_unique_address]] Hash hasher;
    [[no_unique_address]] Pred pred;
    [[no_unique_address]] Allocator alloc;
    Bucket *buckets;
    // ... public:
    // ...
};

Here, hasher, pred, and alloc could have the same address as buckets if their respective types are all empty. — end example]
10 Modules

10.1 Module units and purviews

module-declaration:
  export-keyword\opt module-keyword module-name module-partition\opt attribute-specifier-seq\opt ;
module-name:
  module-name-qualifier\opt identifier
module-partition:
  : module-name-qualifier\opt identifier
module-name-qualifier:
  identifier .
  module-name-qualifier identifier .

1 A module unit is a translation unit that contains a module-declaration. A named module is the collection of module units with the same module-name. The identifiers module and import shall not appear as identifiers in a module-name or module-partition. All module-names either beginning with an identifier consisting of std followed by zero or more digits or containing a reserved identifier (5.10) are reserved and shall not be specified in a module-declaration; no diagnostic is required. If any identifier in a reserved module-name is a reserved identifier, the module name is reserved for use by C++ implementations; otherwise it is reserved for future standardization. The optional attribute-specifier-seq appertains to the module-declaration.

2 A module interface unit is a module unit whose module-declaration starts with export-keyword; any other module unit is a module implementation unit. A named module shall contain exactly one module interface unit with no module-partition, known as the primary module interface unit of the module; no diagnostic is required.

3 A module partition is a module unit whose module-declaration contains a module-partition. A named module shall not contain multiple module partitions with the same module-partition. All module partitions of a module that are module interface units shall be directly or indirectly exported by the primary module interface unit (10.3). No diagnostic is required for a violation of these rules.

[Note 1: Module partitions can be imported only by other module units in the same module. The division of a module into module units is not visible outside the module. — end note]

4 [Example 1:

Translation unit #1:
  export module A;
  export import :Foo;
  export int baz();

Translation unit #2:
  export module A:Foo;
  import :Internals;
  export int foo() { return 2 * (bar() + 1); }

Translation unit #3:
  module A:Internals;
  int bar();

Translation unit #4:
  module A;
  import :Internals;
  int bar() { return baz() - 10; }
  int baz() { return 30; }

Module A contains four translation units:

(4.1) — a primary module interface unit,
(4.2) — a module partition A:Foo, which is a module interface unit forming part of the interface of module A,
(4.3) — a module partition A:Internals, which does not contribute to the external interface of module A, and

§ 10.1
(4.4) — a module implementation unit providing a definition of bar and baz, which cannot be imported because it does not have a partition name.

— end example]

5 A module unit purview is the sequence of tokens starting at the module-declaration and extending to the end of the translation unit. The purview of a named module M is the set of module unit purviews of M’s module units.

6 The global module is the collection of all global-module-fragments and all translation units that are not module units. Declarations appearing in such a context are said to be in the purview of the global module.

[Note 2: The global module has no name, no module interface unit, and is not introduced by any module-declaration. — end note]

7 A module is either a named module or the global module. A declaration is attached to a module as follows:

(7.1) — If the declaration

(7.1.1) — is a replaceable global allocation or deallocation function (17.6.3.2, 17.6.3.3), or

(7.1.2) — is a namespace-definition with external linkage, or

(7.1.3) — appears within a linkage-specification,

it is attached to the global module.

(7.2) — Otherwise, the declaration is attached to the module in whose purview it appears.

8 A module-declaration that contains neither an export-keyword nor a module-partition implicitly imports the primary module interface unit of the module as if by a module-import-declaration.

[Example 2:
Translation unit #1:
module B:Y;
int y();
// does not implicitly import B
Translation unit #2:
export module B;
import :Y;
int n = y();
// OK, does not create interface dependency cycle
Translation unit #3:
module B:X1;
int &a = n;
// error: n not visible here
Translation unit #4:
module B:X2;
import B;
int &b = n;
// OK
Translation unit #5:
module B;
int &c = n;
// implicitly imports B
// OK
— end example]

10.2 Export declaration [module.interface]

eexport-declaration:
export declaration
export { declaration-seqopt }
export-keyword module-import-declaration

1 An export-declaration shall appear only at namespace scope and only in the purview of a module interface unit. An export-declaration shall not appear directly or indirectly within an unnamed namespace or a private-module-fragment. An export-declaration has the declarative effects of its declaration, declaration-seq (if any), or module-import-declaration. An export-declaration does not establish a scope and its declaration or declaration-seq shall not contain an export-declaration or module-import-declaration.

2 A declaration is exported if it is

(2.1) — a namespace-scope declaration declared within an export-declaration, or

§ 10.2
(2.2) — a namespace-definition that contains an exported declaration, or
(2.3) — a declaration within a header unit (10.3) that introduces at least one name.

A exported declaration that is not a module-import-declaration shall declare at least one name. If the declaration is not within a header unit, it shall not declare a name with internal linkage.

[Example 1:

Source file "a.h":
   export int x;

Translation unit #1:
   module;
   #include "a.h"       // error: declaration of x is not in the
   // purview of a module interface unit
   export module M;
   export namespace {}   // error: does not introduce any names
   export namespace {
      int a1;           // error: export of name with internal linkage
   }
   namespace {
      export int a2;     // error: export of name with internal linkage
   }
   export static int b;  // error: b explicitly declared static
   export int f();      // OK
   export namespace N { }
   export using namespace N;  // error: does not declare a name

— end example]

If the declaration is a using-declaration (9.9) and is not within a header unit, all entities to which all of the using-declarators ultimately refer (if any) shall have been introduced with a name having external linkage.

[Example 2:

Source file "b.h":
   int f();

Importable header "c.h":
   int g();

Translation unit #1:
   export module X;
   export int h();

Translation unit #2:
   module;
   #include "b.h"
   export module M;
   import "c.h";
   import X;
   export using ::f, ::g, ::h;    // OK
   struct S;
   export using ::S;             // error: S has module linkage
   namespace N {
      export int h();
      static int h(int);         // #1
   }
   export using N::h;            // error: #1 has internal linkage

— end example]

[Note 1: These constraints do not apply to type names introduced by typedef declarations and alias-declarations.

[Example 3:

   export module M;
   struct S;
   export using T = S;          // OK, exports name T denoting type S

§ 10.2
A redeclaration of an exported declaration of an entity is implicitly exported. An exported redeclaration of a non-exported declaration of an entity is ill-formed.

Example 4:
```
export module M;
struct S { int n; };
typedef S S;  // OK, does not redeclare an entity
export typedef S S;  // error: exported declaration follows non-exported declaration
```

A name is exported by a module if it is introduced or redeclared by an exported declaration in the purview of that module.

Note 2: Exported names have either external linkage or no linkage; see 6.6. Namespace-scope names exported by a module are visible to name lookup in any translation unit importing that module; see 6.4.6. Class and enumeration member names are visible to name lookup in any context in which a definition of the type is reachable. — end note

Example 5:
```
Interface unit of M:
export module M;
export struct X {
    static void f();
    struct Y {};
};
namespace {
    struct S {};
}
export void f(S);  // OK
struct T {};
export T id(T);  // OK
export struct A;  // A exported as incomplete
export auto rootFinder(double a) {
    return [=](double x) { return (x + a/x)/2; };
}
export const int n = 5;  // OK, n has external linkage

Implementation unit of M:
module M;
struct A {
    int value;
};

Main program:
import M;
int main() {
    X::f();  // OK, X is exported and definition of X is reachable
    X::Y y;  // OK, X::Y is exported as a complete type
    auto f = rootFinder(2);  // OK
    return A(45).value;  // error: A is incomplete
}
```

Note 3: Redeclaring a name in an export-declaration cannot change the linkage of the name (6.6).

Example 6:

Interface unit of M:
```
export module M;
```

§ 10.2
static int f(); // #1
export int f(); // error: #1 gives internal linkage
struct S; // #2
export struct S; // error: #2 gives module linkage
namespace {
    namespace N {
        extern int x; // #3
    }
}
export int N::x; // error: #3 gives internal linkage
— end example]
— end note]

9 [Note 4: Declarations in an exported namespace-definition or in an exported linkage-specification (9.11) are exported and subject to the rules of exported declarations.

[Example 7:
export module M;
export namespace N {
    int x; // OK
    static_assert(1 == 1); // error: does not declare a name
}
— end example]
— end note]

10.3 Import declaration [module.import]

module-import-declaration:
    import-keyword module-name attribute-specifier-seq_opt;
    import-keyword module-partition attribute-specifier-seq_opt;
    import-keyword header-name attribute-specifier-seq_opt;

1 A module-import-declaration shall only appear at global namespace scope. In a module unit, all module-import-declarations and export-declarations exporting module-import-declarations shall precede all other declarations in the declaration-seq of the translation-unit and of the private-module-fragment (if any). The optional attribute-specifier-seq opt pertains to the module-import-declaration.

2 A module-import-declaration imports a set of translation units determined as described below.

[Note 1: Namespace-scope names exported by the imported translation units become visible (6.4.6) in the importing translation unit and declarations within the imported translation units become reachable (10.7) in the importing translation unit after the import declaration. — end note]

3 A module-import-declaration that specifies a module-name M imports all module interface units of M.

4 A module-import-declaration that specifies a module-partition shall only appear after the module-declaration in a module unit of some module M. Such a declaration imports the so-named module partition of M.

5 A module-import-declaration that specifies a header-name H imports a synthesized header unit, which is a translation unit formed by applying phases 1 to 7 of translation (5.2) to the source file or header nominated by H, which shall not contain a module-declaration.

[Note 2: All declarations within a header unit are implicitly exported (10.2), and are attached to the global module (10.1). — end note]

An importable header is a member of an implementation-defined set of headers that includes all importable C++ library headers (16.4.2.3). H shall identify an importable header. Given two such module-import-declarations:

(5.1) — if their header-names identify different headers or source files (15.3), they import distinct header units;
(5.2) — otherwise, if they appear in the same translation unit, they import the same header unit;
(5.3) — otherwise, it is unspecified whether they import the same header unit.

[Note 3: It is therefore possible that multiple copies exist of entities declared with internal linkage in an importable header. — end note]

[Note 4: A module-import-declaration nominating a header-name is also recognized by the preprocessor, and results in macros defined at the end of phase 4 of translation of the header unit being made visible as described in 15.5. Any other module-import-declaration does not make macros visible. — end note]
A declaration of a name with internal linkage is permitted within a header unit despite all declarations being implicitly exported (10.2).

[Note 5: A definition that appears in multiple translation units cannot in general refer to such names (6.3). — end note]

A header unit shall not contain a definition of a non-inline function or variable whose name has external linkage.

When a module-import-declaration imports a translation unit \( T \), it also imports all translation units imported by exported module-import-declarations in \( T \); such translation units are said to be exported by \( T \). Additionally, when a module-import-declaration in a module unit of some module \( M \) imports another module unit \( U \) of \( M \), it also imports all translation units imported by non-exported module-import-declarations in the module unit purview of \( U \). These rules can in turn lead to the importation of yet more translation units.

A module implementation unit shall not be exported.

[Example 1:
Translation unit #1:

```
module M:Part;
```

Translation unit #2:

```
export module M;
export import :Part; // error: exported partition :Part is an implementation unit
```

—end example]

A module implementation unit of a module \( M \) that is not a module partition shall not contain a module-import-declaration nominating \( M \).

[Example 2:
```
module M;
import M;
// error: cannot import M in its own unit
```

—end example]

A translation unit has an interface dependency on a translation unit \( U \) if it contains a declaration (possibly a module-declaration) that imports \( U \) or if it has an interface dependency on a translation unit that has an interface dependency on \( U \). A translation unit shall not have an interface dependency on itself.

[Example 3:
```
Interface unit of M1:
export module M1;
import M2;

Interface unit of M2:
export module M2;
import M3;

Interface unit of M3:
export module M3;
import M1; // error: cyclic interface dependency M3 \( \rightarrow \) M1 \( \rightarrow \) M2 \( \rightarrow \) M3
```

—end example]

### 10.4 Global module fragment

```
global-module-fragment:
  module-keyword ; declaration-seq_opt
```

[module.global.frag]

[Note 1: Prior to phase 4 of translation, only preprocessing directives can appear in the declaration-seq (15.1). — end note]

A global-module-fragment specifies the contents of the global module fragment for a module unit. The global module fragment can be used to provide declarations that are attached to the global module and usable within the module unit.

102) This is consistent with the rules for visibility of imported names (6.4.6).
A declaration $D$ is decl-reachable from a declaration $S$ in the same translation unit if:

1. $D$ does not declare a function or function template and $S$ contains an id-expression, namespace-name, type-name, template-name, or concept-name naming $D$, or
2. $D$ declares a function or function template that is named by an expression (6.3) appearing in $S$, or
3. $S$ contains an expression $E$ of the form
   \[ \text{postfix-expression (expression-listopt) } \]
   whose postfix-expression denotes a dependent name, or for an operator expression whose operator denotes a dependent name, and $D$ is found by name lookup for the corresponding name in an expression synthesized from $E$ by replacing each type-dependent argument or operand with a value of a placeholder type with no associated namespaces or entities, or
4. $S$ contains an expression that takes the address of an overloaded function (12.5) whose set of overloads contains $D$ and for which the target type is dependent, or
5. there exists a declaration $M$ that is not a namespace-definition for which $M$ is decl-reachable from $S$ and either
   1. $D$ is decl-reachable from $M$, or
   2. $D$ redeclares the entity declared by $M$ or $M$ redeclares the entity declared by $D$, and $D$ is neither a friend declaration nor a block-scope declaration, or
   3. $D$ declares a namespace $N$ and $M$ is a member of $N$, or
   4. one of $M$ and $D$ declares a class or class template $C$ and the other declares a member or friend of $C$, or
   5. one of $D$ and $M$ declares an enumeration $E$ and the other declares an enumerator of $E$, or
   6. $D$ declares a function or variable and $M$ is declared in $D$, or
   7. one of $M$ and $D$ declares a template and the other declares a partial or explicit specialization or an implicit or explicit instantiation of that template, or
   8. one of $M$ and $D$ declares a class or enumeration type and the other introduces a typedef name for linkage purposes for that type.

In this determination, it is unspecified

- whether a reference to an alias-declaration, typedef declaration, using-declaration, or namespace-alias-definition is replaced by the declarations they name prior to this determination,
- whether a simple-template-id that does not denote a dependent type and whose template-name names an alias template is replaced by its denoted type prior to this determination,
- whether a decltype-specifier that does not denote a dependent type is replaced by its denoted type prior to this determination, and
- whether a non-value-dependent constant expression is replaced by the result of constant evaluation prior to this determination.

4 A declaration $D$ in a global module fragment of a module unit is discarded if $D$ is not decl-reachable from any declaration in the declaration-seq of the translation-unit.

[Note 2: A discarded declaration is neither reachable nor visible to name lookup outside the module unit, nor in template instantiations whose points of instantiation (13.8.5.1) are outside the module unit, even when the instantiation context (10.6) includes the module unit. — end note]

5 [Example 1:

```c
const int size = 2;
int ary1[size]; // unspecified whether size is decl-reachable from ary1
constexpr int identity(int x) { return x; }
int ary2[identity(2)]; // unspecified whether identity is decl-reachable from ary2

template<typename> struct S;
template<typename, int> struct S2;
constexpr int g(int);
```

103) A declaration can appear within a lambda-expression in the initializer of a variable.
template<typename T, int N>  
S<S2<T, g(N)>> f();  // S, S2, g, and :: are decl-reachable from f

template<int N>  
void h() noexcept(g(N) == N);  // g and :: are decl-reachable from h

[Example 2:

Source file "foo.h":

```cpp
namespace N {
  struct X {}
  int d();
  int e();
  inline int f(X, int = d()) { return e(); }
  int g(X);
  int h(X);
}
```

Module M interface:

```cpp
module;
#include "foo.h"
export module M;
template<typename T> int use_f() {
  N::X x;  // N::X, N, and :: are decl-reachable from use_f
  return f(x, 123);  // N::f is decl-reachable from use_f,
                     // N::e is indirectly decl-reachable from use_f
                     // because it is decl-reachable from N::f, and
                     // N::d is decl-reachable from use_f
                     // because it is decl-reachable from N::f
                     // even though it is not used in this call
}

template<typename T> int use_g() {
  N::X x;  // N::X, N, and :: are decl-reachable from use_g
  return g((T(), x));  // N::g is not decl-reachable from use_g
}

template<typename T> int use_h() {
  N::X x;  // N::X, N, and :: are decl-reachable from use_h
  return h((T(), x));  // N::h is not decl-reachable from use_h, but
                       // N::h is decl-reachable from use_h<int>
}
int k = use_h<int>();  // use_h<int> is decl-reachable from k, so
                        // N::h is decl-reachable from k
```

Module M implementation:

```cpp
module M;
int a = use_f<int>();  // OK
int b = use_g<int>();  // error: no viable function for call to g;
                       // g is not decl-reachable from purview of
                       // module M's interface, so is discarded
int c = use_h<int>();  // OK
```

— end example]

10.5 Private module fragment [module.private.frag]

```
private-module-fragment:
  module-keyword : private ; declaration-seq_opt
```

A `private-module-fragment` shall appear only in a primary module interface unit (10.1). A module unit with a `private-module-fragment` shall be the only module unit of its module; no diagnostic is required.

[Note 1: A `private-module-fragment` ends the portion of the module interface unit that can affect the behavior of other translation units. A `private-module-fragment` allows a module to be represented as a single translation unit without making all of the contents of the module reachable to importers. The presence of a `private-module-fragment` affects:
(2.1) — the point by which the definition of an exported inline function is required (9.2.8),
(2.2) — the point by which the definition of an exported function with a placeholder return type is required (9.2.9.6),
(2.3) — whether a declaration is required not to be an exposure (6.6),
(2.4) — where definitions for inline functions and templates must appear (6.3, 9.2.8, 13.1),
(2.5) — the instantiation contexts of templates instantiated before it (10.6), and
(2.6) — the reachability of declarations within it (10.7).

—end note

3 [Example 1:

```c
export module A;
export inline void fn_e();  // error: exported inline function fn_e not defined
// before private module fragment
inline void fn_m();         // OK, module-linkage inline function
static void fn_s();
export struct X;
export void g(X *x) {
    fn_s();              // OK, call to static function in same translation unit
    fn_m();              // OK, call to module-linkage inline function
}
export X *factory();        // OK
module :private;
struct X {};
X *factory() {              // definition not reachable from importers of A
    return new X();
}
void fn_e() {}
void fn_m() {}
void fn_s() {}
—end example]
```

10.6 Instantiation context

The instantiation context is a set of points within the program that determines which names are visible to argument-dependent name lookup (6.5.3) and which declarations are reachable (10.7) in the context of a particular declaration or template instantiation.

2 During the implicit definition of a defaulted function (11.4.4, 11.11.1), the instantiation context is the union of the instantiation context from the definition of the class and the instantiation context of the program construct that resulted in the implicit definition of the defaulted function.

3 During the implicit instantiation of a template whose point of instantiation is specified as that of an enclosing specialization (13.8.5.1), the instantiation context is the union of the instantiation context of the enclosing specialization and, if the template is defined in a module interface unit of a module \( M \) and the point of instantiation is not in a module interface unit of \( M \), the point at the end of the `declaration-seq` of the primary module interface unit of \( M \) (prior to the `private-module-fragment`, if any).

4 During the implicit instantiation of a template that is implicitly instantiated because it is referenced from within the implicit definition of a defaulted function, the instantiation context is the instantiation context of the defaulted function.

5 During the instantiation of any other template specialization, the instantiation context comprises the point of instantiation of the template.

6 In any other case, the instantiation context at a point within the program comprises that point.

[Example 1:

Translation unit #1:

```c
export module stuff;
export template<typename T, typename U> void foo(T, U u) { auto v = u; }
export template<typename T, typename U> void bar(T, U u) { auto v = *u; }
```
Translation unit #2:

```cpp
export module M1;
import "defn.h";  // provides struct X {};
import stuff;
export template<typename T> void f(T t) {
    X x;
    foo(t, x);
}
```

Translation unit #3:

```cpp
export module M2;
import "decl.h";  // provides struct X; (not a definition)
import stuff;
export template<typename T> void g(T t) {
    X *x;
    bar(t, x);
}
```

Translation unit #4:

```cpp
import M1;
import M2;
void test() {
    f(0);
    g(0);
}
```

The call to `f(0)` is valid; the instantiation context of `foo<int, X>` comprises

1. the point at the end of translation unit #1,
2. the point at the end of translation unit #2, and
3. the point of the call to `f(0),`

so the definition of `X` is reachable (10.7).

It is unspecified whether the call to `g(0)` is valid: the instantiation context of `bar<int, X>` comprises

1. the point at the end of translation unit #1,
2. the point at the end of translation unit #3, and
3. the point of the call to `g(0),`

so the definition of `X` need not be reachable, as described in 10.7. — end example

10.7 Reachability

A translation unit `U` is necessarily reachable from a point `P` if `U` is a module interface unit on which the translation unit containing `P` has an interface dependency, or the translation unit containing `P` imports `U`, in either case prior to `P` (10.3).

[Note 1: While module interface units are reachable even when they are only transitively imported via a non-exported import declaration, namespace-scope names from such module interface units are not visible to name lookup (6.4.6). — end note]

All translation units that are necessarily reachable are reachable. Additional translation units on which the point within the program has an interface dependency may be considered reachable, but it is unspecified which are and under what circumstances.104

[Note 2: It is advisable to avoid depending on the reachability of any additional translation units in programs intending to be portable. — end note]

A declaration `D` is reachable if, for any point `P` in the instantiation context (10.6),

1. `D` appears prior to `P` in the same translation unit, or
2. `D` is not discarded (10.4), appears in a translation unit that is reachable from `P`, and does not appear within a private-module-fragment.

---

104) Implementations are therefore not required to prevent the semantic effects of additional translation units involved in the compilation from being observed.
The accumulated properties of all reachable declarations of an entity within a context determine the behavior of the entity within that context.

[Note 4: These reachable semantic properties include type completeness, type definitions, initializers, default arguments of functions or template declarations, attributes, visibility of class or enumeration member names to ordinary lookup, etc. Since default arguments are evaluated in the context of the call expression, the reachable semantic properties of the corresponding parameter types apply in that context.]

[Example 1:

Translation unit #1:
```plaintext
export module M:A;
export struct B;
```

Translation unit #2:
```plaintext
module M:B;
struct B {
  operator int();
};
```

Translation unit #3:
```plaintext
module M:C;
import :A;
B b1;
```

Translation unit #4:
```plaintext
export module M;
export import :A;
import :B;
B b2;
export void f(B b = B());
```

Translation unit #5:
```plaintext
module X;
import M;
B b3;
void g() { f(); }
```

—end example]

—end note]

[Note 5: An entity can have reachable declarations even if it is not visible to name lookup. — end note]

[Example 2:

Translation unit #1:
```plaintext
export module A;
struct X {};
export using Y = X;
```

Translation unit #2:
```plaintext
module B;
import A;
Y y;  // OK, definition of X is reachable
X x;  // error: X not visible to unqualified lookup
```

—end example]
11 Classes

11.1 Preamble

A class is a type. Its name becomes a class-name (11.3) within its scope.

```
class-name:
  identifier
  simple-template-id
```

A class-specifier or an elaborated-type-specifier (9.2.9.4) is used to make a class-name. An object of a class consists of a (possibly empty) sequence of members and base class objects.

```
class-specifier:
  class-head { member-specification opt }
class-head:
  class-key attribute-specifier-seq opt class-head-name class-virt-specifier opt base-clause opt
class-head-name:
  nested-name-specifier opt class-name
class-virt-specifier:
  final
class-key:
  class
  struct
  union
```

A class declaration where the class-name in the class-head-name is a simple-template-id shall be an explicit specialization (13.9.4) or a partial specialization (13.7.6). A class-specifier whose class-head omits the class-head-name defines an unnamed class.

[Note 1: An unnamed class thus can’t be final. — end note]

2 A class-name is inserted into the scope in which it is declared immediately after the class-name is seen. The class-name is also inserted into the scope of the class itself; this is known as the injected-class-name. For purposes of access checking, the injected-class-name is treated as if it were a public member name. A class-specifier is commonly referred to as a class definition. A class is considered defined after the closing brace of its class-specifier has been seen even though its member functions are in general not yet defined. The optional attribute-specifier-seq appertains to the class; the attributes in the attribute-specifier-seq are thereafter considered attributes of the class whenever it is named.

3 If a class-head-name contains a nested-name-specifier, the class-specifier shall refer to a class that was previously declared directly in the class or namespace to which the nested-name-specifier refers, or in an element of the inline namespace set (9.8.2) of that namespace (i.e., not merely inherited or introduced by a using-declaration), and the class-specifier shall appear in a namespace enclosing the previous declaration. In such cases, the nested-name-specifier of the class-head-name of the definition shall not begin with a decltype-specifier.

[Note 2: The class-key determines whether the class is a union (11.5) and whether access is public or private by default (11.9). A union holds the value of at most one data member at a time. — end note]

4 If a class is marked with the class-virt-specifier final and it appears as a class-or-decltype in a base-clause (11.7), the program is ill-formed. Whenever a class-key is followed by a class-head-name, the identifier final, and a colon or left brace, final is interpreted as a class-virt-specifier.

[Example 1:
```
struct A;
struct A final {}; // OK: definition of struct A.
// not value-initialization of variable final

struct X {
  struct C { constexpr operator int() { return 5; } ;
  struct B final : C{}; // OK: definition of nested class B,
```
11.2 Properties of classes

1 A **trivially copyable class** is a class:

- that has at least one eligible copy constructor, move constructor, copy assignment operator, or move assignment operator (11.4.4, 11.4.5.3, 11.4.6),
- where each eligible copy constructor, move constructor, copy assignment operator, and move assignment operator is trivial, and
- that has a trivial, non-deleted destructor (11.4.7).

2 A **trivial class** is a class that is trivially copyable and has one or more eligible default constructors (11.4.5.2), all of which are trivial.

3 A class $S$ is a **standard-layout class** if it:

- has no non-static data members of type non-standard-layout class (or array of such types) or reference,
- has no virtual functions (11.7.3) and no virtual base classes (11.7.2),
- has the same access control (11.9) for all non-static data members,
- has no non-standard-layout base classes,
- has at most one base class subobject of any given type,
- has all non-static data members and bit-fields in the class and its base classes first declared in the same class, and
- has no element of the set $M(S)$ of types as a base class, where for any type $X$, $M(X)$ is defined as follows: $^{105}$

- If $X$ is a non-union class type with no (possibly inherited (11.7)) non-static data members, the set $M(X)$ is empty.
- If $X$ is a non-union class type with a non-static data member of type $X_0$ that is either of zero size or is the first non-static data member of $X$ (where said member may be an anonymous union), the set $M(X)$ consists of $X_0$ and the elements of $M(X_0)$.
- If $X$ is a union type, the set $M(X)$ is the union of all $M(U_i)$ and the set containing all $U_i$, where each $U_i$ is the type of the $i$th non-static data member of $X$.
- If $X$ is an array type with element type $X_e$, the set $M(X)$ consists of $X_e$ and the elements of $M(X_e)$.
- If $X$ is a non-class, non-array type, the set $M(X)$ is empty.

4 **Example 1:**

```c
struct B { int i; };
// standard-layout class
struct C : B { };
// standard-layout class
struct D : C { };
// standard-layout class
struct E : D { char : 4; };
// not a standard-layout class
```

$^{105}$ This ensures that two subobjects that have the same class type and that belong to the same most derived object are not allocated at the same address (7.6.10).
struct Q {};  
struct S : Q {};  
struct T : Q {};  
struct U : S, T {};  // not a standard-layout class  
  —end example

5 A standard-layout struct is a standard-layout class defined with the class-key struct or the class-key class. A standard-layout union is a standard-layout class defined with the class-key union.

6 [Note 3: Standard-layout classes are useful for communicating with code written in other programming languages. Their layout is specified in 11.4. — end note]

7 [Example 2:  
  struct N {  // neither trivial nor standard-layout
    int i;
    int j;
    virtual ~N();
  };

  struct T {  // trivial but not standard-layout
    int i;
    private:
    int j;
  };

  struct SL {  // standard-layout but not trivial
    int i;
    int j;
    ~SL();
  };

  struct POD {  // both trivial and standard-layout
    int i;
    int j;
  };
  —end example]

8 [Note 4: Aggregates of class type are described in 9.4.2. — end note]

9 A class S is an implicit-lifetime class if it is an aggregate or has at least one trivial eligible constructor and a trivial, non-deleted destructor.

11.3 Class names

A class definition introduces a new type.

[Example 1:  
  struct X { int a; };  
  struct Y { int a; };  
  X a1;
  Y a2;
  int a3;

declares three variables of three different types. This implies that

  a1 = a2;  // error: Y assigned to X
  a1 = a3;  // error: int assigned to X

are type mismatches, and that

  int f(X);
  int f(Y);

declare an overloaded (Clause 12) function f() and not simply a single function f() twice. For the same reason,

  struct S { int a; };  
  struct S { int a; };  // error: double definition

is ill-formed because it defines S twice. — end example]
A class declaration introduces the class name into the scope where it is declared and hides any class, variable, function, or other declaration of that name in an enclosing scope (6.4). If a class name is declared in a scope where a variable, function, or enumerator of the same name is also declared, then when both declarations are in scope, the class can be referred to only using an elaborated-type-specifier (6.5.5).

**Example 2:**

```c
struct stat {
    // ...
};

stat gstat;    // use plain stat to define variable
int stat(struct stat*);    // redeclare stat as function

void f() {
    struct stat* ps;    // struct prefix needed to name struct stat
    stat(ps);            // call stat()
}
```

A declaration consisting solely of class-key identifier; is either a redeclaration of the name in the current scope or a forward declaration of the identifier as a class name. It introduces the class name into the current scope.

**Example 3:**

```c
struct s { int a; };

void g() {
    struct s;    // hide global struct s with a block-scope declaration
    s* p;    // refer to local struct s
    struct s { char* p; };    // define local struct s
    struct s;    // redeclaration, has no effect
}
```

**Note 1:** Such declarations allow definition of classes that refer to each other.

**Example 4:**

```c
class Vector;

class Matrix {
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};

class Vector {
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};
```

Declaration of friends is described in 11.9.4, operator functions in 12.6. — end example]

— end note]

**Example 5:**

```c
struct s { int a; };

void g(int s) {
    struct s* p = new struct s;    // global s
    p->a = s;    // parameter s
}
```

— end example]
4 [Note 3: The declaration of a class name takes effect immediately after the identifier is seen in the class definition or elaborated-type-specifier. For example,

```c
class A * A;
```

first specifies A to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form class A must be used to refer to the class. Such artistry with names can be confusing and is best avoided. — end note]

5 A simple-template-id is only a class-name if its template-name names a class template.

11.4 Class members

11.4.1 General

```c
member-specification:
  member-declaration member-specification_opt
access-specifier : member-specification_opt

member-declaration:
  attribute-specifier-seq_opt decl-specifier-seq_opt member-declarator-list_opt ;
  function-definition
  using-declaration
  using-enumeration
  static_assert-declaration
  explicit-specialization
  deduction-guide
  alias-declaration
  opaque-enumeration
  empty-declaration

member-declarator-list:
  member-declarator
  member-declarator-list , member-declarator

member-declarator:
  declarator virt-specifier-seq_opt pure-specifier_opt
  declarator requires-clause
  declarator brace-or-equal-initializer_opt
  identifier_opt attribute-specifier-seq_opt : constant-expression brace-or-equal-initializer_opt

virt-specifier-seq:
  virt-specifier
  virt-specifier-seq virt-specifier

virt-specifier:
  override
  final

pure-specifier:
  = 0
```

1 The member-specification in a class definition declares the full set of members of the class; no member can be added elsewhere. A direct member of a class X is a member of X that was first declared within the member-specification of X, including anonymous union objects (11.5.2) and direct members thereof. Members of a class are data members, member functions (11.4.2), nested types, enumerators, and member templates (13.7.3) and specializations thereof.

[Note 1: A specialization of a static data member template is a static data member. A specialization of a member function template is a member function. A specialization of a member class template is a nested class. — end note]

2 A member-declaration does not declare new members of the class if it is

(2.1) — a friend declaration (11.9.4),
(2.2) — a deduction-guide (13.7.2.3),
(2.3) — a template-declaration whose declaration is one of the above,
(2.4) — a static_assert-declaration,
(2.5) — a using-declaration (9.9), or
(2.6) — an empty-declaration.
For any other member-declaration, each declared entity that is not an unnamed bit-field (11.4.10) is a member of the class, and each such member-declaration shall either declare at least one member name of the class or declare at least one unnamed bit-field.

3 A data member is a non-function member introduced by a member-declarator. A member function is a member that is a function. Nested types are classes (11.3, 11.4.11) and enumerations (9.7.1) declared in the class and arbitrary types declared as members by use of a typedef declaration (9.2.4) or alias-declaration. The enumerators of an unscoped enumeration (9.7.1) defined in the class are members of the class.

4 A data member or member function may be declared static in its member-declaration, in which case it is a static member (see 11.4.9) (a static data member (11.4.9.3) or static member function (11.4.9.2), respectively) of the class. Any other data member or member function is a non-static member (a non-static data member or non-static member function (11.4.3), respectively).

[Note 2: A non-static data member of non-reference type is a member subobject of a class object (6.7.2). — end note]

5 A member shall not be declared twice in the member-specification, except that

(5.1) — a nested class or member class template can be declared and then later defined, and

(5.2) — an enumeration can be introduced with an opaque-enum-declaration and later redeclared with an enum-specifier.

[Note 3: A single name can denote several member functions provided their types are sufficiently different (12.2). — end note]

6 A complete-class context of a class is a

(6.1) — function body (9.5.1),
(6.2) — default argument (9.3.4.7),
(6.3) — noexcept-specifier (14.5), or
(6.4) — default member initializer

within the member-specification of the class.

[Note 4: A complete-class context of a nested class is also a complete-class context of any enclosing class, if the nested class is defined within the member-specification of the enclosing class. — end note]

7 A class is considered a completely-defined object type (6.8) (or complete type) at the closing} of the class-specifier. The class is regarded as complete within its complete-class contexts; otherwise it is regarded as incomplete within its own class member-specification.

8 In a member-declarator, an equals immediately following the declarator is interpreted as introducing a pure-specifier if the declarator-id has function type, otherwise it is interpreted as introducing a brace-or-equal-initializer.

[Example 1:

```c
struct S {
    using T = void();
    T * p = 0;        // OK: brace-or-equal-initializer
    virtual T f = 0;  // OK: pure-specifier
};
— end example]
```

9 In a member-declarator for a bit-field, the constant-expression is parsed as the longest sequence of tokens that could syntactically form a constant-expression.

[Example 2:

```c
int a;
const int b = 0;
struct S {
    int x1 : 8 = 42;       // OK, "= 42" is brace-or-equal-initializer
    int x2 : 8 { 42 };    // OK, "{ 42 \}" is brace-or-equal-initializer
    int y1 : true ? 8 : a = 42; // OK, brace-or-equal-initializer is absent
    int y2 : true ? 8 : b = 42; // error: cannot assign to const int
    int y3 : (true ? 8 : b) = 42; // OK, "= 42" is brace-or-equal-initializer
    int z : 1 || new int { 0 }; // OK, brace-or-equal-initializer is absent
};
— end example]
```
A brace-or-equal-initializer shall appear only in the declaration of a data member. (For static data members, see 11.4.9.3; for non-static data members, see 11.10.3 and 9.4.2). A brace-or-equal-initializer for a non-static data member specifies a default member initializer for the member, and shall not directly or indirectly cause the implicit definition of a defaulted default constructor for the enclosing class or the exception specification of that constructor.

A member shall not be declared with the extern storage-class-specifier. Within a class definition, a member shall not be declared with the thread_local storage-class-specifier unless also declared static.

The decl-specifier-seq may be omitted in constructor, destructor, and conversion function declarations only; when declaring another kind of member the decl-specifier-seq shall contain a type-specifier that is not a cv-qualifier. The member-declarator-list can be omitted only after a class-specifier or an enum-specifier or in a friend declaration (11.9.4).

The optional attribute-specifier-seq in a member-declaration appertains to each of the entities declared by the member-declarators; it shall not appear if the optional member-declarator-list is omitted.

A virt-specifier-seq shall contain at most one of each virt-specifier. A virt-specifier-seq shall appear only in the first declaration of a virtual member function (11.7.3).

The type of a non-static data member shall not be an incomplete type (6.8), an abstract class type (11.7.4), or a (possibly multi-dimensional) array thereof.

Note 5: In particular, a class C cannot contain a non-static member of class C, but it can contain a pointer or reference to an object of class C. — end note

Note 6: See 7.5.4 for restrictions on the use of non-static data members and non-static member functions. — end note

Note 7: The type of a non-static member function is an ordinary function type, and the type of a non-static data member is an ordinary object type. There are no special member function types or data member types. — end note

Example 3: A simple example of a class definition is

```c
struct tnode {
    char tword[20];
    int count;
    tnode* left;
    tnode* right;
};
```

which contains an array of twenty characters, an integer, and two pointers to objects of the same type. Once this definition has been given, the declaration

```c
tnode s, *sp;
```

declares s to be a tnode and sp to be a pointer to a tnode. With these declarations, sp->count refers to the count member of the object to which sp points; s.left refers to the left subtree pointer of the object s; and s.right->tword[0] refers to the initial character of the tword member of the right subtree of s. — end example

Note 8: Non-static data members of a (non-union) class with the same access control (11.9) and non-zero size (6.7.2) are allocated so that later members have higher addresses within a class object (7.6.9). The order of allocation of non-static data members with different access control is unspecified. Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions (11.7.3) and virtual base classes (11.7.2). — end note

If T is the name of a class, then each of the following shall have a name different from T:

(20.1) every static data member of class T;
(20.2) every member function of class T;
(20.3) every member of class T that is itself a type;
(20.4) every member template of class T;
(20.5) every enumerator of every member of class T that is an unscoped enumerated type; and
(20.6) every member of every anonymous union that is a member of class T.

Note 9: This restriction does not apply to constructors, which do not have names (11.4.5) — end note

In addition, if class T has a user-declared constructor (11.4.5), every non-static data member of class T shall have a name different from T.
The common initial sequence of two standard-layout struct (11.2) types is the longest sequence of non-static data members and bit-fields in declaration order, starting with the first such entity in each of the structs, such that corresponding entities have layout-compatible types, either both entities are declared with the no_unique_address attribute (9.12.10) or neither is, and either both entities are bit-fields with the same width or neither is a bit-field.

[Example 4:

```c
struct A { int a; char b; }
struct B { const int b1; volatile char b2; }
struct C { int c; unsigned : 0; char b; }
struct D { int d; char b : 4; }
struct E { unsigned int e; char b; }
```

The common initial sequence of `A` and `B` comprises all members of either class. The common initial sequence of `A` and `C` and of `A` and `D` comprises the first member in each case. The common initial sequence of `A` and `E` is empty. — end example]

Two standard-layout struct (11.2) types are layout-compatible classes if their common initial sequence comprises all members and bit-fields of both classes (6.8).

Two standard-layout unions are layout-compatible if they have the same number of non-static data members and corresponding non-static data members (in any order) have layout-compatible types (6.8).

In a standard-layout union with an active member (11.5) of struct type `T1`, it is permitted to read a non-static data member `m` of another union member of struct type `T2` provided `m` is part of the common initial sequence of `T1` and `T2`; the behavior is as if the corresponding member of `T1` were nominated.

[Example 5:

```c
struct T1 { int a, b; }
struct T2 { int c; double d; }
union U { T1 t1; T2 t2; }
int f() {
    U u = { { 1, 2 } }; // active member is t1
    return u.t2.c; // OK, as if u.t1.a were nominated
}
```

— end example]

[Note 10: Reading a volatile object through a glvalue of non-volatile type has undefined behavior (9.2.9.2). — end note]

If a standard-layout class object has any non-static data members, its address is the same as the address of its first non-static data member if that member is not a bit-field. Its address is also the same as the address of each of its base class subobjects.

[Note 11: There might therefore be unnamed padding within a standard-layout struct object inserted by an implementation, but not at its beginning, as necessary to achieve appropriate alignment. — end note]

[Note 12: The object and its first subobject are pointer-interconvertible (6.8.3, 7.6.1.9). — end note]

### 11.4.2 Member functions

A member function may be defined (9.5) in its class definition, in which case it is an inline (9.2.8) member function if it is attached to the global module, or it may be defined outside of its class definition if it has already been declared but not defined in its class definition.

[Note 1: A member function is also inline if it is declared inline, constexpr, or consteval. — end note]

A member function definition that appears outside of the class definition shall appear in a namespace scope enclosing the class definition. Except for member function definitions that appear outside of a class definition, and except for explicit specializations of member functions of class templates and member function templates (13.9) appearing outside of the class definition, a member function shall not be redeclared.

[Note 2: There can be at most one definition of a non-inline member function in a program. There can be more than one inline member function definition in a program. See 6.3 and 9.2.8. — end note]

[Note 3: Member functions of a class have the linkage of the name of the class. See 6.6. — end note]

If the definition of a member function is lexically outside its class definition, the member function name shall be qualified by its class name using the :: operator.
[Note 4: A name used in a member function definition (that is, in the parameter-declaration-clause including the default arguments (9.3.4.7) or in the member function body) is looked up as described in 6.5. — end note]

[Example 1:
  struct X {
    typedef int T;
    static T count;
    void f(T);
  };
  void X::f(T t = count) { }
  The member function f of class X is defined in global scope; the notation X::f specifies that the function f is a member of class X and in the scope of class X. In the function definition, the parameter type T refers to the typedef member T declared in class X and the default argument count refers to the static data member count declared in class X. — end example]

4 [Note 5: A static local variable or local type in a member function always refers to the same entity, whether or not the member function is inline. — end note]

Previously declared member functions may be mentioned in friend declarations.

Member functions of a local class shall be defined inline in their class definition, if they are defined at all.

7 [Note 6: A member function can be declared (but not defined) using a typedef for a function type. The resulting member function has exactly the same type as it would have if the function declarator were provided explicitly, see 9.3.4.6. For example,
  typedef void fv();
  typedef void fvc() const;
  struct S {
    fv memfunc1;  // equivalent to: void memfunc1();
    void memfunc2();
    fvc memfunc3;  // equivalent to: void memfunc3() const;
  };
  fv S::* pmfv1 = &S::memfunc1;
  fvc S::* pmfv2 = &S::memfunc2;
  fvc S::* pmfv3 = &S::memfunc3;
  Also see 13.4. — end note]

11.4.3 Non-static member functions [class.mfct.non-static]

11.4.3.1 General [class.mfct.non-static.general]

1 A non-static member function may be called for an object of its class type, or for an object of a class derived (11.7) from its class type, using the class member access syntax (7.6.1.5, 12.4.2.2). A non-static member function may also be called directly using the function call syntax (7.6.1.3, 12.4.2.2) from within its class or a class derived from its class, or a member thereof, as described below.

2 If a non-static member function of a class X is called for an object that is not of type X, or of a type derived from X, the behavior is undefined.

3 When an id-expression (7.5.4) that is not part of a class member access syntax (7.6.1.5) and not used to form a pointer to member (7.6.2.2) is used in a member of class X in a context where this can be used (7.5.2), if name lookup (6.5) resolves the name in the id-expression to a non-static non-type member of some class C, and if either the id-expression is potentially evaluated or C is X or a base class of X, the id-expression is transformed into a class member access expression (7.6.1.5) using (*this) (11.4.3.2) as the postfix-expression to the left of the . operator.

[Note 1: If C is not X or a base class of X, the class member access expression is ill-formed. — end note]

This transformation does not apply in the template definition context (13.8.3.2).

[Example 1:
  struct tnode {
    char tword[20];
    int count;
    tnode* left;
    tnode* right;
    void set(const char*, tnode* l, tnode* r);
  };

§ 11.4.3.1 267
void tnode::set(const char* w, tnode* l, tnode* r) {
    count = strlen(w)+1;
    if (sizeof(tword)<=count)
        perror("tnode string too long");
    strcpy(tword,w);
    left = l;
    right = r;
}

void f(tnode n1, tnode n2) {
    n1.set("abc", &n2, 0);
    n2.set("def", 0, 0);
}

In the body of the member function tnode::set, the member names tword, count, left, and right refer to members of the object for which the function is called. Thus, in the call n1.set("abc", &n2, 0), tword refers to n1.tword, and in the call n2.set("def", 0, 0), it refers to n2.tword. The functions strlen, perror, and strcpy are not members of the class tnode and should be declared elsewhere.

A non-static member function may be declared const, volatile, or const volatile. These cv-qualifiers affect the type of the this pointer (11.4.3.2). They also affect the function type (9.3.4.6) of the member function; a member function declared const is a const member function, a member function declared volatile is a volatile member function and a member function declared const volatile is a const volatile member function.

[Example 2:

```c
struct X {
    void g() const;
    void h() const volatile;
};
```

X::g is a const member function and X::h is a const volatile member function. — end example] 4

A non-static member function may be declared with a ref-qualifier (9.3.4.6); see 12.4.2.
5

A non-static member function may be declared virtual (11.7.3) or pure virtual (11.7.4).

11.4.3.2 The this pointer

[class.this]

In the body of a non-static (11.4.2) member function, the keyword this is a prvalue whose value is a pointer to the object for which the function is called. The type of this in a member function whose type has a cv-qualifier-seq cv and whose class is X is “pointer to cv X”.

[Note 1: Thus in a const member function, the object for which the function is called is accessed through a const access path. — end note]

[Example 1:

```c
struct s {
    int a;
    int f() const;
    int g() { return a++; }
    int h() const { return a++; } // error
};
```

int s::f() const { return a; }

The a++ in the body of s::h is ill-formed because it tries to modify (a part of) the object for which s::h() is called. This is not allowed in a const member function because this is a pointer to const; that is, *this has const type. — end example] 2

[Note 2: Similarly, volatile semantics (9.2.9.2) apply in volatile member functions when accessing the object and its non-static data members. — end note]

3

A member function whose type has a cv-qualifier-seq cv1 can be called on an object expression (7.6.1.5) of type cv2 T only if cv1 is the same as or more cv-qualified than cv2 (6.8.4).

[Example 2:

```c
106)
```

3

See, for example, <cstring> (21.5.3).
void k(s& x, const s& y) {
    x.f();
    x.g();
    y.f();
    y.g();         // error
}

The call y.g() is ill-formed because y is const and s::g() is a non-const member function, that is, s::g() is
less-qualified than the object expression y. —end example]

4 [Note 3: Constructors and destructors cannot be declared const, volatile, or const volatile. However, these
functions can be invoked to create and destroy objects with cv-qualified types; see 11.4.5 and 11.4.7. —end note]

11.4.4 Special member functions [special]

Default constructors (11.4.5.2), copy constructors, move constructors (11.4.5.3), copy assignment operators,
move assignment operators (11.4.6), and prospective destructors (11.4.7) are special member functions.

[Note 1: The implementation will implicitly declare these member functions for some class types when the program
does not explicitly declare them. The implementation will implicitly define them if they are odr-used (6.3) or needed
for constant evaluation (7.7). —end note]

An implicitly-declared special member function is declared at the closing } of the class-specifier. Programs
shall not define implicitly-declared special member functions.

2 Programs may explicitly refer to implicitly-declared special member functions.

[Example 1: A program may explicitly call or form a pointer to member to an implicitly-declared special member
function.

struct A { };                     // implicitly declared A::operator=
struct B : A {
    B& operator=(const B &);  
};
B& B::operator=(const B& s) {
    this->A::operator=(s);     // well-formed
    return *this;
}
—end example]

3 [Note 2: The special member functions affect the way objects of class type are created, copied, moved, and destroyed,
and how values can be converted to values of other types. Often such special member functions are called implicitly.
—end note]

Special member functions obey the usual access rules (11.9).

[Example 2: Declaring a constructor protected ensures that only derived classes and friends can create objects using
it. —end example]

5 Two special member functions are of the same kind if:

(5.1) — they are both default constructors,

(5.2) — they are both copy or move constructors with the same first parameter type, or

(5.3) — they are both copy or move assignment operators with the same first parameter type and the same
cv-qualifiers and ref-qualifier, if any.

6 An eligible special member function is a special member function for which:

(6.1) — the function is not deleted,

(6.2) — the associated constraints (13.5), if any, are satisfied, and

(6.3) — no special member function of the same kind is more constrained (13.5.5).

7 For a class, its non-static data members, its non-virtual direct base classes, and, if the class is not abstract
(11.7.4), its virtual base classes are called its potentially constructed subobjects.

8 A defaulted special member function is constexpr-compatible if the corresponding implicitly-declared special
member function would be a constexpr function.
### 11.4.5 Constructors [class.ctor]

#### 11.4.5.1 General [class.ctor.general]

1. A **constructor** is introduced by a declaration whose **declarator** is a function declarator (9.3.4.6) of the form

   \[
   \text{ptr-declarator}\ (\ \text{parameter-declaration-clause}\ )\ \text{nothrow-specifier}_{\text{opt}}\ \text{attribute-specifier-seq}_{\text{opt}}
   \]

   where the **ptr-declarator** consists solely of an **id-expression**, an optional **attribute-specifier-seq**, and optional surrounding parentheses, and the **id-expression** has one of the following forms:

   - **(1.1)** — in a member-declaration that belongs to the member-specification of a class or class template but is not a friend declaration (11.9.4), the **id-expression** is the injected-class-name (11.1) of the immediately-enclosing entity or
   - **(1.2)** — in a declaration at namespace scope or in a friend declaration, the **id-expression** is a **qualified-id** that names a constructor (6.5.4.2).

Constructors do not have names. In a constructor declaration, each **decl-specifier** in the optional **decl-specifier-seq** shall be **friend**, **inline**, **constexpr**, or an **explicit-specifier**.

**Example 1:**

```cpp
struct S {
    S(); // declares the constructor
};
S::S() { } // defines the constructor
```

2. A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (7.6.1.4) will cause a constructor to be called to initialize an object.

   **[Note 1]:** The syntax looks like an explicit call of the constructor. — *end note*

**Example 2:**

```cpp
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
```

   **[Note 2]:** For initialization of objects of class type see 11.10. — *end note*

3. An object created in this way is unnamed.

   **[Note 3]:** 6.7.7 describes the lifetime of temporary objects. — *end note*

   **[Note 4]:** Explicit constructor calls do not yield lvalues, see 7.2.1. — *end note*

4. A constructor can be invoked for a **const**, **volatile** or **const volatile** object. **const** and **volatile** semantics (9.2.9.2) are not applied on an object under construction. They come into effect when the constructor for the most derived object (6.7.2) ends.

5. A **return** statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.

6. A constructor shall not be a coroutine.

#### 11.4.5.2 Default constructors [class.default.ctor]

1. A **default constructor** for a class **X** is a constructor of class **X** for which each parameter that is not a function parameter pack has a default argument (including the case of a constructor with no parameters). If there is no user-declared constructor for class **X**, a non-explicit constructor having no parameters is implicitly declared as defaulted (9.5). An implicitly-declared default constructor is an inline public member of its class.

2. A defaulted default constructor for class **X** is defined as deleted if:

   **(2.1)** — **X** is a union that has a variant member with a non-trivial default constructor and no variant member of **X** has a default member initializer,
X is a non-union class that has a variant member $M$ with a non-trivial default constructor and no variant member of the anonymous union containing $M$ has a default member initializer,

any non-static data member with no default member initializer (11.4) is of reference type,

any non-variant non-static data member of const-qualified type (or array thereof) with no brace-or-equal-initializer is not const-default-constructible (9.4),

X is a union and all of its variant members are of const-qualified type (or array thereof),

X is a non-union class and all members of any anonymous union member are of const-qualified type (or array thereof),

any potentially constructed subobject, except for a non-static data member with a brace-or-equal-initializer, has class type $M$ (or array thereof) and either $M$ has no default constructor or overload resolution (12.4) as applied to find $M$'s corresponding constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor, or

any potentially constructed subobject has a type with a destructor that is deleted or inaccessible from the defaulted default constructor.

A default constructor is trivial if it is not user-provided and if:

- its class has no virtual functions (11.7.3) and no virtual base classes (11.7.2), and
- no non-static data member of its class has a default member initializer (11.4), and
- all the direct base classes of its class have trivial default constructors, and
- for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

Otherwise, the default constructor is non-trivial.

A default constructor that is defaulted and not defined as deleted is implicitly defined when it is odr-used (6.3) to create an object of its class type (6.7.2), when it is needed for constant evaluation (7.7), or when it is explicitly defaulted after its first declaration. The implicitly-defined default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with no ctor-initializer (11.10.3) and an empty compound-statement. If that user-written default constructor would be ill-formed, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a constexpr constructor (9.2.6), the implicitly-defined default constructor is constexpr. Before the defaulted default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members are implicitly defined.

Default constructors are called implicitly to create class objects of static, thread, or automatic storage duration (6.7.5.2, 6.7.5.3, 6.7.5.4) defined without an initializer (9.4), are called to create class objects of dynamic storage duration (6.7.5.5) created by a new-expression in which the new-initializer is omitted (7.6.2.8), or are called when the explicit type conversion syntax (7.6.1.4) is used. A program is ill-formed if the default constructor for an object is implicitly used and the constructor is not accessible (11.9).

11.4.5.3 Copy/move constructors

A non-template constructor for class $X$ is a copy constructor if its first parameter is of type $X&$, const $X&$, volatile $X&$ or const volatile $X&$, and either there are no other parameters or else all other parameters have default arguments (9.3.4.7).

[Example 1: $X::X(const X&)$ and $X::X(X&, int = 1)$ are copy constructors.

```c
struct X {
    X(int);
    X(const X&, int = 1);
};
X a(1); // calls X(int);
X b(a, 0); // calls X(const X&, int);
X c = b;  // calls X(const X&, int);
```
A non-template constructor for class \( X \) is a move constructor if its first parameter is of type \( X&&, \) \( \text{const } X&&, \) \( \text{volatile } X&&, \) or \( \text{const volatile } X&&, \) and either there are no other parameters or else all other parameters have default arguments (9.3.4.7).

**Example 2:** \( Y::Y(Y&&) \) is a move constructor.

```cpp
struct Y {
  Y(const Y&);
  Y(Y&&);
};
extern Y f(int);
Y d(f(1)); // calls Y(Y&&)
Y e = d;   // calls Y(const Y&)
```

—end example]  

[Note 1: All forms of copy/move constructor can be declared for a class.

**Example 3:**

```cpp
struct X {
  X(const X&);
  X(X&);  // OK
  X(X&&); // OK, but possibly not sensible
};
```

—end example]

—end note

[Note 2: If a class \( X \) only has a copy constructor with a parameter of type \( X& \), an initializer of type \( \text{const } X \) or \( \text{volatile } X \) cannot initialize an object of type \( \text{cv } X \).]

**Example 4:**

```cpp
struct X {
  X();                            // default constructor
  X(X&&);                        // copy constructor with a non-const parameter
};
const X cx;
X x = cx;                       // error: \( X::X(X&&) \) cannot copy \( cx \) into \( x \)
```

—end example]

—end note

A declaration of a constructor for a class \( X \) is ill-formed if its first parameter is of type \( \text{cv } X \) and either there are no other parameters or else all other parameters have default arguments. A member function template is never instantiated to produce such a constructor signature.

**Example 5:**

```cpp
struct S {
  template<typename T> S(T);
  S();
};

S g;

void h() {
  S a(g); // does not instantiate the member template to produce \( S::S<S>(S) \);
  // uses the implicitly declared copy constructor
}
```

—end example]

If the class definition does not explicitly declare a copy constructor, a non-explicit one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (9.5). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor (D.9).

The implicitly-declared copy constructor for a class \( X \) will have the form
X::X(const X&)  
if each potentially constructed subobject of a class type M (or array thereof) has a copy constructor whose first parameter is of type const M& or const volatile M&.\(^{107}\) Otherwise, the implicitly-declared copy constructor will have the form  
X::X(X&)  

If the definition of a class X does not explicitly declare a move constructor, a non-explicit one will be implicitly declared as defaulted if and only if

- (8.1) X does not have a user-declared copy constructor,
- (8.2) X does not have a user-declared copy assignment operator,
- (8.3) X does not have a user-declared move assignment operator, and
- (8.4) X does not have a user-declared destructor.

[Note 3: When the move constructor is not implicitly declared or explicitly supplied, expressions that otherwise would have invoked the move constructor might instead invoke a copy constructor. — end note]

The implicitly-declared move constructor for class X will have the form  
X::X(X&&)

An implicitly-declared copy/move constructor is an inline public member of its class. A defaulted copy/move constructor for a class X is defined as deleted (9.5.3) if X has:

- (10.1) a potentially constructed subobject type M (or array thereof) that cannot be copied/moved because overload resolution (12.4), as applied to find M’s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor,
- (10.2) a variant member whose corresponding constructor as selected by overload resolution is non-trivial,
- (10.3) any potentially constructed subobject of a type with a destructor that is deleted or inaccessible from the defaulted constructor, or,
- (10.4) for the copy constructor, a non-static data member of rvalue reference type.

[Note 4: A defaulted move constructor that is defined as deleted is ignored by overload resolution (12.4, 12.5). Such a constructor would otherwise interfere with initialization from an rvalue which can use the copy constructor instead. — end note]

A copy/move constructor for class X is trivial if it is not user-provided and if:

- (11.1) class X has no virtual functions (11.7.3) and no virtual base classes (11.7.2), and
- (11.2) the constructor selected to copy/move each direct base class subobject is trivial, and
- (11.3) for each non-static data member of X that is of class type (or array thereof), the constructor selected to copy/move that member is trivial; otherwise the copy/move constructor is non-trivial.

A copy/move constructor that is defaulted and not defined as deleted is implicitly defined when it is odr-used (6.3), when it is needed for constant evaluation (7.7), or when it is explicitly defaulted after its first declaration.

[Note 5: The copy/move constructor is implicitly defined even if the implementation elided its odr-use (6.3, 6.7.7). — end note]

If the implicitly-defined constructor would satisfy the requirements of a constexpr constructor (9.2.6), the implicitly-defined constructor is constexpr.

Before the defaulted copy/move constructor for a class is implicitly defined, all non-user-provided copy/move constructors for its potentially constructed subobjects are implicitly defined.

[Note 6: An implicitly-declared copy/move constructor has an implied exception specification (14.5). — end note]

The implicitly-defined copy/move constructor for a non-union class X performs a memberwise copy/move of its bases and members.

[Note 7: Default member initializers of non-static data members are ignored. See also the example in 11.10.3. — end note]

107\) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a volatile lvalue; see C.5.7.
The order of initialization is the same as the order of initialization of bases and members in a user-defined constructor (see 11.10.3). Let \( x \) be either the parameter of the constructor or, for the move constructor, an \( x \)value referring to the parameter. Each base or non-static data member is copied/moved in the manner appropriate to its type:

1. If the member is an array, each element is direct-initialized with the corresponding subobject of \( x \);
2. If a member \( m \) has \( \text{rvalue} \) reference type \( T&& \), it is direct-initialized with \( \text{static
cast\( T&& \)(x.m)} \);
3. Otherwise, the base or member is direct-initialized with the corresponding base or member of \( x \).

Virtual base class subobjects shall be initialized only once by the implicitly-defined copy/move constructor (see 11.10.3).

The implicitly-defined copy/move constructor for a union \( X \) copies the object representation (6.8) of \( X \). For each object nested within (6.7.2) the object that is the source of the copy, a corresponding object \( o \) nested within the destination is identified (if the object is a subobject) or created (otherwise), and the lifetime of \( o \) begins before the copy is performed.

### 11.4.6 Copy/move assignment operator

A user-declared copy assignment operator \( X::\text{operator=} \) is a non-static non-template member function of class \( X \) with exactly one parameter of type \( X, X&, \text{const } X&, \text{volatile } X&, \text{or } \text{const volatile } X& \).

[Note 1: An overloaded assignment operator must be declared to have only one parameter; see 12.6.3.2. — end note]

[Note 2: More than one form of copy assignment operator can be declared for a class. — end note]

[Note 3: If a class \( X \) only has a copy assignment operator with a parameter of type \( X& \), an expression of type const \( X \) cannot be assigned to an object of type \( X \).

[Example 1:]
```cpp
struct X {
    X();
    X& operator=(X&);
};
const X cx;
X x;
void f() {
    x = cx;  // error: X::operator=(X&) cannot assign cx into x
}
```

[—end example]

[—end note]

2. If the class definition does not explicitly declare a copy assignment operator, one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy assignment operator is defined as deleted; otherwise, it is defined as defaulted (9.5). The latter case is deprecated if the class has a user-declared copy constructor or a user-declared destructor (D.9). The implicitly-declared copy assignment operator for a class \( X \) will have the form

\[ X& X::\text{operator}=(\text{const } X&) \]

if

- each direct base class \( B \) of \( X \) has a copy assignment operator whose parameter is of type \( \text{const } B&, \text{const volatile } B&, \text{or } B, \text{and} \)
- for all the non-static data members of \( X \) that are of a class type \( M \) (or array thereof), each such class type has a copy assignment operator whose parameter is of type \( \text{const } M&, \text{const volatile } M&, \text{or } M \).

Otherwise, the implicitly-declared copy assignment operator will have the form

\[ X& X::\text{operator}=(X&) \]

108) Because a template assignment operator or an assignment operator taking an rvalue reference parameter is never a copy assignment operator, the presence of such an assignment operator does not suppress the implicit declaration of a copy assignment operator. Such assignment operators participate in overload resolution with other assignment operators, including copy assignment operators, and, if selected, will be used to assign an object.

109) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a \text{volatile} lvalue; see C.5.7.
A user-declared move assignment operator \(X::\text{operator=}\) is a non-static non-template member function of class \(X\) with exactly one parameter of type \(X&&\), \(\text{const } X&&\), \(\text{volatile } X&&\), or \(\text{const volatile } X&&\).

[Note 4: An overloaded assignment operator must be declared to have only one parameter; see 12.6.3.2. — end note]
[Note 5: More than one form of move assignment operator can be declared for a class. — end note]

If the definition of a class \(X\) does not explicitly declare a move assignment operator, one will be implicitly declared as defaulted if and only if

(4.1) \(X\) does not have a user-declared copy constructor,
(4.2) \(X\) does not have a user-declared move constructor,
(4.3) \(X\) does not have a user-declared copy assignment operator, and
(4.4) \(X\) does not have a user-declared destructor.

[Example 2: The class definition

\[
\text{struct S} \\
\text{int a;} \\
S& \text{operator=(const S&) }= \text{default};
\]

will not have a default move assignment operator implicitly declared because the copy assignment operator has been user-declared. The move assignment operator may be explicitly defaulted.

\[
\text{struct S} \\
\text{int a;} \\
S& \text{operator=(const S&) }= \text{default}; \\
S& \text{operator=(S&) }= \text{default};
\]

— end example]

The implicitly-declared move assignment operator for a class \(X\) will have the form

\[
X& \ X::\text{operator=(X&&)}
\]

The implicitly-declared copy/move assignment operator for class \(X\) has the return type \(X&\); it returns the object for which the assignment operator is invoked, that is, the object assigned to. An implicitly-declared copy/move assignment operator is an inline public member of its class.

A defaulted copy/move assignment operator for class \(X\) is defined as deleted if \(X\) has:

(7.1) a variant member with a non-trivial corresponding assignment operator and \(X\) is a union-like class, or
(7.2) a non-static data member of \text{const} non-class type (or array thereof), or
(7.3) a non-static data member of reference type, or
(7.4) a direct non-static data member of class type \(M\) (or array thereof) or a direct base class \(M\) that cannot be copied/moved because overload resolution (12.4), as applied to find \(M\)'s corresponding assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the defaulted assignment operator.

[Note 6: A defaulted move assignment operator that is defined as deleted is ignored by overload resolution (12.4, 12.5). — end note]

Because a copy/move assignment operator is implicitly declared for a class if not declared by the user, a base class copy/move assignment operator is always hidden by the corresponding assignment operator of a derived class (12.6.3.2). A \textit{using-declaration} (9.9) that brings in from a base class an assignment operator with a parameter type that could be that of a copy/move assignment operator for the derived class is not considered an explicit declaration of such an operator and does not suppress the implicit declaration of the derived class operator; the operator introduced by the \textit{using-declaration} is hidden by the implicitly-declared operator in the derived class.

A copy/move assignment operator for class \(X\) is trivial if it is not user-provided and if:

(9.1) \(X\) has no virtual functions (11.7.3) and no virtual base classes (11.7.2), and
(9.2) the assignment operator selected to copy/move each direct base class subobject is trivial, and
(9.3) for each non-static data member of \(X\) that is of class type (or array thereof), the assignment operator selected to copy/move that member is trivial;

§ 11.4.6 275
otherwise the copy/move assignment operator is non-trivial.

10 A copy/move assignment operator for a class X that is defaulted and not defined as deleted is implicitly defined when it is odr-used (6.3) (e.g., when it is selected by overload resolution to assign to an object of its class type), when it is needed for constant evaluation (7.7), or when it is explicitly defaulted after its first declaration. The implicitly-defined copy/move assignment operator is constexpr if

\begin{align}
& \text{(10.1)} \quad X \text{ is a literal type, and} \\
& \text{(10.2)} \quad \text{the assignment operator selected to copy/move each direct base class subobject is a constexpr function, and} \\
& \text{(10.3)} \quad \text{for each non-static data member of } X \text{ that is of class type (or array thereof), the assignment operator selected to copy/move that member is a constexpr function.}
\end{align}

11 Before the defaulted copy/move assignment operator for a class is implicitly defined, all non-user-provided copy/move assignment operators for its direct base classes and its non-static data members are implicitly defined.

[Note 7: An implicitly-declared copy/move assignment operator has an implied exception specification (14.5). — end note]

12 The implicitly-defined copy/move assignment operator for a non-union class X performs memberwise copy/move assignment of its subobjects. The direct base classes of X are assigned first, in the order of their declaration in the base-specifier-list, and then the immediate non-static data members of X are assigned, in the order in which they were declared in the class definition. Let x be either the parameter of the function or, for the move operator, an xvalue referring to the parameter. Each subobject is assigned in the manner appropriate to its type:

\begin{align}
& \text{(12.1)} \quad \text{if the subobject is of class type, as if by a call to } \text{operator=} \text{ with the subobject as the object expression and the corresponding subobject of } x \text{ as a single function argument (as if by explicit qualification; that is, ignoring any possible virtual overriding functions in more derived classes);} \\
& \text{(12.2)} \quad \text{if the subobject is an array, each element is assigned, in the manner appropriate to the element type;} \\
& \text{(12.3)} \quad \text{if the subobject is of scalar type, the built-in assignment operator is used.}
\end{align}

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the implicitly-defined copy/move assignment operator.

[Example 3:

\begin{verbatim}
struct V { }
struct A : virtual V { }
struct B : virtual V { }
struct C : B, A { }
\end{verbatim}

It is unspecified whether the virtual base class subobject V is assigned twice by the implicitly-defined copy/move assignment operator for C. — end example]

13 The implicitly-defined copy assignment operator for a union X copies the object representation (6.8) of X. If the source and destination of the assignment are not the same object, then for each object nested within (6.7.2) the object that is the source of the copy, a corresponding object o nested within the destination is created, and the lifetime of o begins before the copy is performed.

### 11.4.7 Destructors [class.dtor]

A prospective destructor is introduced by a declaration whose declarator is a function declarator (9.3.4.6) of the form

\begin{verbatim}
ptr-declarator ( parameter-declaration-clause ) noexcept-specifier_opt attribute-specifier-seq_opt
\end{verbatim}

where the ptr-declarator consists solely of an id-expression, an optional attribute-specifier-seq, and optional surrounding parentheses, and the id-expression has one of the following forms:

\begin{align}
& \text{(1.1)} \quad \text{in a member-declaration that belongs to the member-specification of a class or class template but is not a friend declaration (11.9.4), the id-expression is } \simclassname \text{ and the class-name is the injected-class-name (11.1) of the immediately-enclosing entity or} \\
& \text{(1.2)} \quad \text{in a declaration at namespace scope or in a friend declaration, the id-expression is nested-name-specifier } \\
& \simclassname \text{ and the class-name names the same class as the nested-name-specifier.}
\end{align}
A prospective destructor shall take no arguments (9.3.4.6). Each *decl-specifier* of the *decl-specifier-seq* of a prospective destructor declaration (if any) shall be *friend*, *inline*, *virtual*, *constexpr*, or *consteval*.

If a class has no user-declared prospective destructor, a prospective destructor is implicitly declared as defaulted (9.5). An implicitly-declared prospective destructor is an inline public member of its class.

An implicitly-declared prospective destructor for a class \( X \) will have the form

\[ \sim X() \]

At the end of the definition of a class, overload resolution is performed among the prospective destructors declared in that class with an empty argument list to select the *destructor* for the class, also known as the *selected destructor*. The program is ill-formed if overload resolution fails. Destructor selection does not constitute a reference to, or odr-use (6.3) of, the selected destructor, and in particular, the selected destructor may be deleted (9.5.3).

The address of a destructor shall not be taken. A destructor can be invoked for a *const*, *volatile* or *const* *volatile* object. *Const* and *volatile* semantics (9.2.9.2) are not applied on an object under destruction. They stop being in effect when the destructor for the most derived object (6.7.2) starts.

A defaulted destructor for a class \( X \) is defined as deleted if:

1. \( X \) is a union-like class that has a variant member with a non-trivial destructor,
2. any potentially constructed subobject has class type \( M \) (or array thereof) and \( M \) has a deleted destructor or a destructor that is inaccessible from the defaulted destructor,
3. or, for a virtual destructor, lookup of the non-array deallocation function results in an ambiguity or in a function that is deleted or inaccessible from the defaulted destructor.

A destructor is trivial if it is not user-provided and if:

1. the destructor is not virtual,
2. all of the direct base classes of its class have trivial destructors, and
3. for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

Otherwise, the destructor is *non-trivial*.

A defaulted destructor is a *constexpr* destructor if it satisfies the requirements for a *constexpr* destructor (9.2.6).

A destructor that is defaulted and not defined as deleted is *implicitly defined* when it is odr-used (6.3) or when it is explicitly defaulted after its first declaration.

Before a defaulted destructor for a class is implicitly defined, all the non-user-provided destructors for its base classes and its non-static data members are implicitly defined.

A prospective destructor can be declared *virtual* (11.7.3) and with a *pure-specifier* (11.7.4). If the destructor of a class is virtual and any objects of that class or any derived class are created in the program, the destructor shall be defined. If a class has a base class with a virtual destructor, its destructor (whether user- or implicitly-declared) is virtual.

A destructor is invoked implicitly

1. for a constructed object with static storage duration (6.7.5.2) at program termination (6.9.3.4),
2. for a constructed object with thread storage duration (6.7.5.3) at thread exit,
— for a constructed object with automatic storage duration (6.7.5.4) when the block in which an object is created exits (8.8),
— for a constructed temporary object when its lifetime ends (7.3.5, 6.7.7). In each case, the context of the invocation is the context of the construction of the object. A destructor may also be invoked implicitly through use of a delete-expression (7.6.2.9) for a constructed object allocated by a new-expression (7.6.2.8); the context of the invocation is the delete-expression.

[Note 4: An array of class type contains several subobjects for each of which the destructor is invoked. — end note]

A destructor can also be invoked explicitly. A destructor is potentially invoked if it is invoked or as specified in 7.6.2.8, 8.7.4, 9.4.2, 11.10.3, and 14.2. A program is ill-formed if a destructor that is potentially invoked is deleted or not accessible from the context of the invocation.

At the point of definition of a virtual destructor (including an implicit definition), the non-array deallocation function is determined as if for the expression delete this appearing in a non-virtual destructor of the destructor’s class (see 7.6.2.9). If the lookup fails or if the deallocation function has a deleted definition (9.5), the program is ill-formed.

[Note 5: This assures that a deallocation function corresponding to the dynamic type of an object is available for the delete-expression (11.12). — end note]

In an explicit destructor call, the destructor is specified by a ~ followed by a type-name or decltype-specifier that denotes the destructor’s class type. The invocation of a destructor is subject to the usual rules for member functions (11.4.2); that is, if the object is not of the destructor’s class type and not of a class derived from the destructor’s class type (including when the destructor is invoked via a null pointer value), the program has undefined behavior.

[Note 6: Invoking delete on a null pointer does not call the destructor; see 7.6.2.9. — end note]

[Example 1:]

```c++
struct B {
  virtual ~B() { }
};
struct D : B {
  ~D() { }
};
D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
  D_object.B::~B(); // calls B’s destructor
  B_ptr->~B(); // calls D’s destructor
  B_ptr->~B_alias(); // calls D’s destructor
  B_ptr->B_alias::~B(); // calls B’s destructor
  B_ptr->B_alias::~B_alias(); // calls B’s destructor
}

// end example]

[Note 7: An explicit destructor call must always be written using a member access operator (7.6.1.5) or a qualified-id (7.5.4.3); in particular, the unary-expression ~X() in a member function is not an explicit destructor call (7.6.2.2). — end note]

[Note 8: Explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific addresses using a placement new-expression. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities. For example,]

```c++
void* operator new(std::size_t, void* p) { return p; }
struct X {
  X(int);
  ~X();
};

void f(X* p);

void g() { // rare, specialized use:
  char* buf = new char[sizeof(X)];
}

§ 11.4.7
X* p = new(buf) X(222);     // use buf[] and initialize
f(p);
p->X::~X();                   // cleanup
}
—end note]

19 Once a destructor is invoked for an object, the object’s lifetime ends; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended (6.7.3).

[Example 2: If the destructor for an object with automatic storage duration is explicitly invoked, and the block is subsequently left in a manner that would ordinarily invoke implicit destruction of the object, the behavior is undefined.
—end example]

20 [Note 9: The notation for explicit call of a destructor can be used for any scalar type name (7.5.4.4). Allowing this makes it possible to write code without having to know if a destructor exists for a given type. For example:

typedef int I;
I* p;
p->I::~I();
—end note]

21 A destructor shall not be a coroutine.

11.4.8 Conversions [class.conv]
11.4.8.1 General [class.conv.general]
1 Type conversions of class objects can be specified by constructors and by conversion functions. These conversions are called user-defined conversions and are used for implicit type conversions (7.3), for initialization (9.4), and for explicit type conversions (7.6.1.4, 7.6.3, 7.6.1.9).

2 User-defined conversions are applied only where they are unambiguous (11.8, 11.4.8.3). Conversions obey the access control rules (11.9). Access control is applied after ambiguity resolution (6.5).

3 [Note 1: See 12.4 for a discussion of the use of conversions in function calls as well as examples below. —end note]

4 At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value.

[Example 1:
   struct X {
     operator int();
   };

   struct Y {
     operator X();
   };

   Y a;
   int b = a;     // error: no viable conversion (a.operator X().operator int() not considered)
   int c = X(a);  // OK: a.operator X().operator int()
—end example]

5 User-defined conversions are used implicitly only if they are unambiguous. A conversion function in a derived class does not hide a conversion function in a base class unless the two functions convert to the same type. Function overload resolution (12.4.4) selects the best conversion function to perform the conversion.

[Example 2:
   struct X {
     operator int();
   };

   struct Y : X {
     operator char();
   };

§ 11.4.8.1
void f(Y& a) {
    if (a) {       // error: ambiguous between X::operator int() and Y::operator char()
    }
} — end example]

11.4.8.2 Conversion by constructor [class.convctor]

1 A constructor that is not explicit (9.2.3) specifies a conversion from the types of its parameters (if any) to
the type of its class. Such a constructor is called a converting constructor.

[Example 1:
struct X {
    X(int);
    X(const char*, int = 0);
    X(int, int);
};

void f(X arg) {
    X a = 1;       // a = X(1)
    X b = "Jessie"; // b = X("Jessie",0)
    a = 2;         // a = X(2)
    f(3);          // f(X(3))
    f({1, 2});     // f(X(1,2))
}
] — end example]

2 [Note 1: An explicit constructor constructs objects just like non-explicit constructors, but does so only where
the direct-initialization syntax (9.4) or where casts (7.6.1.9, 7.6.3) are explicitly used; see also 12.4.2.5. A default
constructor can be an explicit constructor; such a constructor will be used to perform default-initialization or value-initialization (9.4).

[Example 2:
struct Z {
    explicit Z();
    explicit Z(int);
    explicit Z(int, int);
};

Z a;         // OK: default-initialization performed
Z b();       // OK: direct initialization syntax used
Z c = {};    // error: copy-list-initialization
Z a1 = 1;    // error: no implicit conversion
Z a3 = Z(1); // OK: direct initialization syntax used
Z a2();     // OK: direct initialization syntax used
Z p = new Z(1); // OK: direct initialization syntax used
Z a4 = (2);  // OK: explicit cast used
Z a5 = static_cast<Z>(1);  // OK: explicit cast used
Z a6 = {3, 4}; // error: no implicit conversion
] — end example]
— end note]

3 A non-explicit copy/move constructor (11.4.5.3) is a converting constructor.
[Note 2: An implicitly-declared copy/move constructor is not an explicit constructor; it can be called for implicit type
conversions. — end note]

11.4.8.3 Conversion functions [class.conv.fct]

1 A member function of a class X having no parameters with a name of the form

    conversion-function-id:
      operator conversion-type-id

    conversion-type-id:
      type-specifier-seq conversion-declarator_opt

    conversion-declarator:
      ptr-operator conversion-declarator_opt
specifies a conversion from X to the type specified by the conversion-type-id. Such functions are called conversion functions. A decl-specifier in the decl-specifier-seq of a conversion function (if any) shall be neither a defining-type-specifier nor static. The type of the conversion function (9.3.4.6) is “function taking no parameter returning conversion-type-id”. A conversion function is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified) same object type (or a reference to it), to a (possibly cv-qualified) base class of that type (or a reference to it), or to cv void.¹¹⁰

[Example 1:]

```c
struct X {
    operator int();
    operator auto() -> short; // error: trailing return type
};

void f(X a) {
    int i = int(a);
    i = (int)a;
    i = a;
}
```

In all three cases the value assigned will be converted by X::operator int(). — end example]

²

A conversion function may be explicit (9.2.3), in which case it is only considered as a user-defined conversion for direct-initialization (9.4). Otherwise, user-defined conversions are not restricted to use in assignments and initializations.

[Example 2:]

```c
class Y { }
struct Z {
    explicit operator Y() const;
};

void h(Z z) {
    Y y1(z);
    // OK: direct-initialization
    Y y2 = z;
    // error: no conversion function candidate for copy-initialization
    Y y3 = (Y)z;
    // OK: cast notation
}
```

void g(X a, X b) {
    int i = (a) ? 1+a : 0;
    int j = (a&&b) ? a+b : i;
    if (a) {
    }
}

— end example]

³

The conversion-type-id shall not represent a function type nor an array type. The conversion-type-id in a conversion-function-id is the longest sequence of tokens that could possibly form a conversion-type-id.

[Note 1: This prevents ambiguities between the declarator operator * and its expression counterparts.

[Example 3:]

```c
&ac.operator int*i; // syntax error:
    // parsed as: &ac.operator int *)i
    // not as: &ac.operator int)*i
```

The * is the pointer declarator and not the multiplication operator. — end example]

This rule also prevents ambiguities for attributes.

[Example 4:]

```c
operator int [[noreturn]] () ; // error: noreturn attribute applied to a type
```

— end example]

¹¹⁰ These conversions are considered as standard conversions for the purposes of overload resolution (12.4.2.2, 12.4.2.5) and therefore initialization (9.4) and explicit casts (7.6.1.9). A conversion to void does not invoke any conversion function (7.6.1.9). Even though never directly called to perform a conversion, such conversion functions can be declared and can potentially be reached through a call to a virtual conversion function in a base class.

§ 11.4.8.3
Conversion functions are inherited.

Conversion functions can be virtual.

A conversion function template shall not have a deduced return type (9.2.9.6).

Example 5:

```cpp
struct S {
    operator auto() const { return 10; } // OK
    template<class T>
    operator auto() const { return 1.2; } // error: conversion function template
};
```

Example 1:

```cpp
struct process {
    static void reschedule();
};
process& g();
void f() {
    process::reschedule(); // OK: no object necessary
    g().reschedule(); // g() is called
}
```

Example 2:

```cpp
int g();
struct X {
    static int g();
};
struct Y : X {
    static int i;
};
int Y::i = g(); // equivalent to Y::g();
```

Static members obey the usual class member access rules (11.9). When used in the declaration of a class member, the static specifier shall only be used in the member declarations that appear within the member-specification of the class definition.

Note 1: It cannot be specified in member declarations that appear in namespace scope. — end note]

11.4.9.2 Static member functions

Note 1: The rules described in 11.4.2 apply to static member functions. — end note]

Note 2: A static member function does not have a this pointer (11.4.3.2). A static member function cannot be declared virtual (9.2.3). There cannot be a static and a non-static member function with the same name, parameter-type-list, and trailing requires-clause (12.2). A static member function cannot not be declared const, volatile, or const volatile (9.3.4.6). — end note]
11.4.9.3 Static data members

A static data member is not part of the subobjects of a class. If a static data member is declared `thread_local` there is one copy of the member per thread. If a static data member is not declared `thread_local` there is one copy of the data member that is shared by all the objects of the class.

A static data member shall not be `mutable` (9.2.2). A static data member shall not be a direct member (11.4) of an unnamed (11.1) or local (11.6) class or of a (possibly indirectly) nested class (11.4.11) thereof.

The declaration of a non-inline static data member in its class definition is not a definition and may be of an incomplete type other than `cv void`. The definition for a static data member that is not defined inline in the class definition shall appear in a namespace scope enclosing the member’s class definition. In the definition at namespace scope, the name of the static data member shall be qualified by its class name using the `::` operator. The `initializer` expression in the definition of a static data member is in the scope of its class (6.4.7).

Example 1:

```cpp
class process {
    static process* run_chain;
    static process* running;
};
```

```cpp
process* process::running = get_main();
process* process::run_chain = running;
```

The static data member `run_chain` of class `process` is defined in global scope; the notation `process::run_chain` specifies that the member `run_chain` is a member of class `process` and in the scope of class `process`. In the static data member definition, the `initializer` expression refers to the static data member `running` of class `process`. — end example

[Note 1: Once the static data member has been defined, it exists even if no objects of its class have been created.]

[Example 2: In the example above, `run_chain` and `running` exist even if no objects of class `process` are created by the program. — end example]

— end note

If a non-volatile non-inline `const` static data member is of integral or enumeration type, its declaration in the class definition can specify a `brace-or-equal-initializer` in which every `initializer-clause` that is an `assignment-expression` is a constant expression (7.7). The member shall still be defined in a namespace scope if it is odr-used (6.3) in the program and the namespace scope definition shall not contain an `initializer`. An inline static data member may be defined in the class definition and may specify a `brace-or-equal-initializer`. If the member is declared with the `constexpr` specifier, it may be redeclared in namespace scope with no `initializer` (this usage is deprecated; see D.7). Declarations of other static data members shall not specify a `brace-or-equal-initializer`.

[Note 2: There is exactly one definition of a static data member that is odr-used (6.3) in a valid program. — end note]

[Note 3: Static data members of a class in namespace scope have the linkage of the name of the class (6.6). — end note]

Static data members are initialized and destroyed exactly like non-local variables (6.9.3.2, 6.9.3.3, 6.9.3.4).

11.4.10 Bit-fields

A `member-declarator` of the form

```cpp
   identifier_opt attribute-specifier-seq_opt : constant-expression brace-or-equal-initializer_opt
```

specifies a bit-field. The optional `attribute-specifier-seq` appertains to the entity being declared. A bit-field shall not be a static member. A bit-field shall have integral or enumeration type; the bit-field semantic property is not part of the type of the class member. The `constant-expression` shall be an integral constant expression with a value greater than or equal to zero and is called the `width` of the bit-field. If the width of a bit-field is larger than the width of the bit-field’s type (or, in case of an enumeration type, of its underlying type), the extra bits are padding bits (6.8). Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit.

[Note 1: Bit-fields straddle allocation units on some machines and not on others. Bit-fields are assigned right-to-left on some machines, left-to-right on others. — end note]
A declaration for a bit-field that omits the identifier declares an unnamed bit-field. Unnamed bit-fields are not members and cannot be initialized. An unnamed bit-field shall not be declared with a cv-qualified type.

[Note 2: An unnamed bit-field is useful for padding to conform to externally-imposed layouts. — end note]

As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary. Only when declaring an unnamed bit-field may the width be zero.

The address-of operator & shall not be applied to a bit-field, so there are no pointers to bit-fields. A non-const reference shall not be bound to a bit-field (9.4.4).

[Note 3: If the initializer for a reference of type const T& is an lvalue that refers to a bit-field, the reference is bound to a temporary initialized to hold the value of the bit-field; the reference is not bound to the bit-field directly. See 9.4.4. — end note]

If a value of integral type (other than bool) is stored into a bit-field of width N and the value would be representable in a hypothetical signed or unsigned integer type with width N and the same signedness as the bit-field’s type, the original value and the value of the bit-field compare equal. If the value true or false is stored into a bit-field of type bool of any size (including a one bit bit-field), the original bool value and the value of the bit-field compare equal. If a value of an enumeration type is stored into a bit-field of the same type and the width is large enough to hold all the values of that enumeration type (9.7.1), the original value and the value of the bit-field compare equal.

[Example 1:

```c
enum BOOL { FALSE=0, TRUE=1 }; 
struct A {
    BOOL b:1;
}; 
A a;
void f() {
    a.b = TRUE; 
    if (a.b == TRUE) // yields true
        { /* ... */ }
} 
—end example]
```

11.4.11 Nested class declarations [class.nest]

A class can be declared within another class. A class declared within another is called a nested class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class.

[Note 1: See 7.5.4 for restrictions on the use of non-static data members and non-static member functions. — end note]

[Example 1:

```c
int x;
int y;
struct enclose {
    int x;
    static int s;

    struct inner {
        void f(int i) {
            int a = sizeof(x); // OK: operand of sizeof is an unevaluated operand
            x = i;    // error: assign to enclose::x
            s = i;    // OK: assign to enclose::s
            ::x = i;  // OK: assign to global x
            y = i;    // OK: assign to global y
        }
        void g(enclose* p, int i) {
            p->x = i;  // OK: assign to enclose::x
        }
    }
};
```
Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class.

Example 2:
```c
struct enclose {
    struct inner {
        static int x;
        void f(int i);
    };

    int enclose::inner::x = 1;

    void enclose::inner::f(int i) { /* ... */ }
};
```

Example 3:
```c
class E {
    class I1; // forward declaration of nested class
    class I2;
    class I1 { };
    class E::I2 { };
};
```

If class X is defined in a namespace scope, a nested class Y may be declared in class X and later defined in the definition of class X or be later defined in a namespace scope enclosing the definition of class X.

Example 3:
```c
class E {
    class I1; // forward declaration of nested class
    class I2;
    class I1 { };
    class E::I2 { };
};
```

Like a member function, a friend function (11.9.4) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (11.4.9), but it has no special access rights to members of an enclosing class.

11.4.12 Nested type names
Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification.

Example 1:
```c
struct X {
    typedef int I;
    class Y { /* ... */ };
    I a;
};
```

```c
I b; // error
Y c; // error
X::Y d; // OK
X::I e; // OK
```

11.5 Unions

1 A union is a class defined with the class-key union.

In a union, a non-static data member is active if its name refers to an object whose lifetime has begun and has not ended (6.7.3). At most one of the non-static data members of an object of union type can be active at any time, that is, the value of at most one of the non-static data members can be stored in a union at any time.

Note 1: One special guarantee is made in order to simplify the use of unions: If a standard-layout union contains several standard-layout structs that share a common initial sequence (11.4), and if a non-static data member of an
object of this standard-layout union type is active and is one of the standard-layout struct members, it is permitted to inspect
the common initial sequence of any of the standard-layout struct members; see 11.4. — end note]

3 The size of a union is sufficient to contain the largest of its non-static data members. Each non-static data
member is allocated as if it were the sole member of a non-union class.

[Note 2: A union object and its non-static data members are pointer-interconvertible (6.8.3, 7.6.1.9). As a consequence,
all non-static data members of a union object have the same address. — end note]

4 A union can have member functions (including constructors and destructors), but it shall not have virtual
(11.7.3) functions. A union shall not have base classes. A union shall not be used as a base class. If a union
contains a non-static data member of reference type the program is ill-formed.

[Note 3: Absent default member initializers (11.4), if any non-static data member of a union has a non-trivial default
constructor (11.4.5.2), copy constructor, move constructor (11.4.5.3), copy assignment operator, move assignment
operator (11.4.6), or destructor (11.4.7), the corresponding member function of the union must be user-provided or it
will be implicitly deleted (9.5.3) for the union. — end note]

5 [Example 1: Consider the following union:

```cpp
union U {
    int i;
    float f;
    std::string s;
};
```

Since `std::string` (21.3) declares non-trivial versions of all of the special member functions, `U` will have an implicitly
deleted default constructor, copy/move constructor, copy/move assignment operator, and destructor. To use `U`, some
or all of these member functions must be user-provided. — end example]

6 When the left operand of an assignment operator involves a member access expression (7.6.1.5) that nominates
a union member, it may begin the lifetime of that union member, as described below. For an expression `E`,
define the set \( S(E) \) of subexpressions of `E` as follows:

\[(6.1) \quad \text{If } E \text{ is of the form } A.B, S(E) \text{ contains the elements of } S(A), \text{ and also contains } A.B \text{ if } B \text{ names a union member of a non-class, non-array type, or of a class type with a trivial default constructor that is not deleted, or an array of such types.}\]

\[(6.2) \quad \text{If } E \text{ is of the form } A[B] \text{ and is interpreted as a built-in array subscripting operator, } S(E) \text{ is } S(A) \text{ if } A \text{ is of array type, } S(B) \text{ if } B \text{ is of array type, and empty otherwise.}\]

\[(6.3) \quad \text{Otherwise, } S(E) \text{ is empty.}\]

In an assignment expression of the form `E1 = E2` that uses either the built-in assignment operator (7.6.19) or
a trivial assignment operator (11.4.6), for each element \( X \) of \( S(E1) \), if modification of \( X \) would have undefined
behavior under 6.7.3, an object of the type of \( X \) is implicitly created in the nominated storage; no initialization
is performed and the beginning of its lifetime is sequenced after the value computation of the left and right
operands and before the assignment.

[Note 4: This ends the lifetime of the previously-active member of the union, if any (6.7.3). — end note]

[Example 2:

```cpp
union A { int x; int y[4]; };  
struct B { A a; };  
union C { B b; int k; };  
int f() {
    C c;               // does not start lifetime of any union member
    c.b.a.y[3] = 4;    // OK: S(c.b.a.y[3]) contains c.b and c.b.a.y;
    // creates objects to hold union members c.b and c.b.a.y
    return c.b.a.y[3]; // OK: c.b.a.y refers to newly created object (see 6.7.3)
}

struct X { const int a; int b; };  
union Y { X x; int k; };  
void g() {
    Y y = {1, 2};      // OK, y.x is active union member (11.4)
    int n = y.x.a;    // undefined behavior: y.x.b modified outside its lifetime,
    y.k = 4;          // OK: ends lifetime of y.x, y.k is active member of union
    y.x.b = n;       // undefined behavior: y.x.b modified outside its lifetime,
}
7  [Note 5: In cases where the above rule does not apply, the active member of a union can only be changed by the use of a placement new-expression. — end note]

[Example 3: Consider an object \( u \) of a \texttt{union} type \( U \) having non-static data members \( m \) of type \( M \) and \( n \) of type \( N \). If \( M \) has a non-trivial destructor and \( N \) has a non-trivial constructor (for instance, if they declare or inherit virtual functions), the active member of \( u \) can be safely switched from \( m \) to \( n \) using the destructor and placement new-expression as follows:

\[
\begin{align*}
\texttt{u.m.}\texttt{~\textasciitilde\texttt{M}();} \\
\texttt{new (\&u.n) N;}
\end{align*}
\]
— end example]

11.5.2 Anonymous unions

A union of the form

\[
\texttt{union \{ member-specification \};}
\]

is called an \textit{anonymous union}; it defines an unnamed type and an unnamed object of that type called an \textit{anonymous union object}. Each \texttt{member-declaration} in the \texttt{member-specification} of an anonymous union shall either define a public non-static data member or be a \texttt{static_assert-declaration}. Nested types, anonymous unions, and functions shall not be declared within an anonymous union. The names of the members of an anonymous union shall be distinct from the names of any other entity in the scope in which the anonymous union is declared. For the purpose of name lookup, after the anonymous union definition, the members of the anonymous union are considered to have been defined in the scope in which the anonymous union is declared.

[Example 1:

\[
\begin{align*}
\texttt{void f() \{} \\
\texttt{\quad \texttt{union \{ int a; const char* p; \};} \\
\texttt{\quad a = 1;} \\
\texttt{\quad p = "Jennifer";} \\
\texttt{\}\}
\end{align*}
\]

Here \( a \) and \( p \) are used like ordinary (non-member) variables, but since they are union members they have the same address. — end example]

Anonymous unions declared in a named namespace or in the global namespace shall be declared \texttt{static}. Anonymous unions declared at block scope shall be declared with any storage class allowed for a block-scope variable, or with no storage class. A storage class is not allowed in a declaration of an anonymous union in a class scope.

[Note 1: A union for which objects, pointers, or references are declared is not an anonymous union.]

[Example 2:

\[
\begin{align*}
\texttt{void f() \{} \\
\texttt{\quad \texttt{union \{ int aa; char* p; \} obj, *ptr = \&obj;} \\
\texttt{\quad aa = 1; \hspace{1cm} // error} \\
\texttt{\quad ptr->aa = 1; \hspace{1cm} // OK} \\
\texttt{\}\}
\end{align*}
\]

The assignment to plain \( aa \) is ill-formed since the member name is not visible outside the union, and even if it were visible, it is not associated with any particular object. — end example]

— end note]

[Note 2: Initialization of unions with no user-declared constructors is described in 9.4.2. — end note]

4  A \textit{union-like class} is a union or a class that has an anonymous union as a direct member. A union-like class \( X \) has a set of \textit{variant members}. If \( X \) is a union, a non-static data member of \( X \) that is not an anonymous union is a variant member of \( X \). In addition, a non-static data member of an anonymous union that is a member of \( X \) is also a variant member of \( X \). At most one variant member of a union may have a default member initializer.

[Example 3:
union U {
    int x = 0;
    union {
        int k;
    };
    union {
        int z;
        int y = 1;  // error: initialization for second variant member of U
    };
};

—end example]

11.6 Local class declarations [class.local]

A class can be declared within a function definition; such a class is called a local class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing function, and has the same access to names outside the function as does the enclosing function.

[Note 1: A declaration in a local class cannot odr-use (6.3) a local entity from an enclosing scope. —end note]

[Example 1:

```c
int x;
void f() {
    static int s;
    int x;
    const int N = 5;
    extern int q();
    int arr[2];
    auto [y, z] = arr;

    struct local {
        int g() { return x; }  // error: odr-use of non-odr-usable variable x
        int h() { return s; }  // OK
        int k() { return ::x; }  // OK
        int l() { return q(); }  // OK
        int m() { return N; }   // OK: not an odr-use
        int* n() { return &N; } // error: odr-use of non-odr-usable variable N
        int p() { return y; }   // error: odr-use of non-odr-usable structured binding y
    };
}
```

local* p = 0;  // error: local not in scope

—end example]

2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (11.9). Member functions of a local class shall be defined within their class definition, if they are defined at all.

3 If class X is a local class a nested class Y may be declared in class X and later defined in the definition of class X or be later defined in the same scope as the definition of class X. A class nested within a local class is a local class.

4 [Note 2: A local class cannot have static data members (11.4.9.3). —end note]

11.7 Derived classes [class.derived]

11.7.1 General [class.derived.general]

A list of base classes can be specified in a class definition using the notation:

```c
base-clause:
    : base-specifier-list

base-specifier-list:
    base-specifier . . . opt
    base-specifier-list , base-specifier . . . opt
```
base-specifier:
  attribute-specifier-seqopt class-or-decltype
  attribute-specifier-seqopt virtual access-specifieropt class-or-decltype
  attribute-specifier-seqopt access-specifier virtualopt class-or-decltype

class-or-decltype:
  nested-name-specifieropt type-name
  nested-name-specifier template simple-template-id
dectype-specifier

access-specifier:
  private
  protected
  public

The optional attribute-specifier-seq appertains to the base-specifier.

2 A class-or-decltype shall denote a (possibly cv-qualified) class type that is not an incompletely defined class (11.4); any cv-qualifiers are ignored. The class denoted by the class-or-decltype of a base-specifier is called a direct base class for the class being defined. During the lookup for a base class name, non-type names are ignored (6.4.10). A class B is a base class of a class D if it is a direct base class of D or a direct base class of one of D’s base classes. A class is an indirect base class of another if it is a base class but not a direct base class. A class is said to be (directly or indirectly) derived from its (direct or indirect) base classes.

[Note 1: See 11.9 for the meaning of access-specifier. — end note]

Unless redeclared in the derived class, members of a base class are also considered to be members of the derived class. Members of a base class other than constructors are said to be inherited by the derived class. Constructors of a base class can also be inherited as described in 9.9. Inherited members can be referred to in expressions in the same manner as other members of the derived class, unless their names are hidden or ambiguous (11.8).

[Note 2: The scope resolution operator :: (7.5.4.3) can be used to refer to a direct or indirect base member explicitly. This allows access to a name that has been redeclared in the derived class. A derived class can itself serve as a base class subject to access control; see 11.9.3. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (7.3.12). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (9.4.4). — end note]

3 The base-specifier-list specifies the type of the base class subobjects contained in an object of the derived class type.

[Example 1:
  struct Base {
    int a, b, c;
  };
  struct Derived : Base {
    int b;
  };
  struct Derived2 : Derived {
    int c;
  };

  Here, an object of class Derived2 will have a subobject of class Derived which in turn will have a subobject of class Base. — end example]

4 A base-specifier followed by an ellipsis is a pack expansion (13.7.4).

5 The order in which the base class subobjects are allocated in the most derived object (6.7.2) is unspecified.

[Note 3: A derived class and its base class subobjects can be represented by a directed acyclic graph (DAG) where an arrow means “directly derived from” (see Figure 2). An arrow need not have a physical representation in memory. A DAG of subobjects is often referred to as a “subobject lattice”. — end note]

6 [Note 4: Initialization of objects representing base classes can be specified in constructors; see 11.10.3. — end note]

7 [Note 5: A base class subobject might have a layout different from the layout of a most derived object of the same type. A base class subobject might have a polymorphic behavior (11.10.5) different from the polymorphic behavior of a most derived object of the same type. A base class subobject can be of zero size; however, two subobjects that have the same class type and that belong to the same most derived object cannot be allocated at the same address (6.7.2). — end note]

§ 11.7.1
11.7.2 Multiple base classes

A class can be derived from any number of base classes.

[Note 1: The use of more than one direct base class is often called multiple inheritance. — end note]

[Example 1:

```cpp
class A { /* ... */ };  
class B { /* ... */ };  
class C { /* ... */ };  
class D : public A, public B, public C { /* ... */ };  
@end example]

[Note 2: The order of derivation is not significant except as specified by the semantics of initialization by constructor (11.10.3), cleanup (11.4.7), and storage layout (11.4, 11.9.2). — end note]

A class shall not be specified as a direct base class of a derived class more than once.

[Note 3: A class can be an indirect base class more than once and can be a direct and an indirect base class. There are limited things that can be done with such a class. The non-static data members and member functions of the direct base class cannot be referred to in the scope of the derived class. However, the static members, enumerations, and types can be unambiguously referred to. — end note]

[Example 2:

```cpp
class X { /* ... */ };  
class Y : public X, public X { /* ... */ };  // error  
class L { public: int next; /* ... */ };  
class A : public L { /* ... */ };  
class B : public L { /* ... */ };  
class C : public A, public B { void f(); /* ... */ };  // well-formed  
class D : public A, public L { void f(); /* ... */ };  // well-formed  
@end example]

A base class specifier that does not contain the keyword virtual specifies a non-virtual base class. A base class specifier that contains the keyword virtual specifies a virtual base class. For each distinct occurrence of a non-virtual base class in the class lattice of the most derived class, the most derived object (6.7.2) shall contain a corresponding distinct base class subobject of that type. For each distinct base class that is specified virtual, the most derived object shall contain a single base class subobject of that type.

[Note 4: For an object of class type C, each distinct occurrence of a (non-virtual) base class L in the class lattice of C corresponds one-to-one with a distinct L subobject within the object of type C. Given the class C defined above, an object of class C will have two subobjects of class L as shown in Figure 3.

[Figure 3: Non-virtual base [fig:class.nonvirt]]
In such lattices, explicit qualification can be used to specify which subobject is meant. The body of function \( C::f \) could refer to the member `next` of each \( L \) subobject:

```cpp
void C::f() { A::next = B::next; } // well-formed
```

Without the `A::` or `B::` qualifiers, the definition of `C::f` above would be ill-formed because of ambiguity (11.8).

---

**Note 5**: In contrast, consider the case with a virtual base class:

```cpp
class V { /* ... */ }
class A : virtual public V { /* ... */ }
class B : virtual public V { /* ... */ }
class C : public A, public B { /* ... */ }
```

![Figure 4: Virtual base](fig:class.virt)

For an object \( c \) of class type \( C \), a single subobject of type \( V \) is shared by every base class subobject of \( c \) that has a virtual base class of type \( V \). Given the class \( C \) defined above, an object of class \( C \) will have one subobject of class \( V \), as shown in Figure 4.

---

**Note 6**: A class can have both virtual and non-virtual base classes of a given type.

```cpp
class B { /* ... */ }
class X : virtual public B { /* ... */ }
class Y : virtual public B { /* ... */ }
class Z : public B { /* ... */ }
class AA : public X, public Y, public Z { /* ... */ }
```

For an object of class \( AA \), all virtual occurrences of base class \( B \) in the class lattice of \( AA \) correspond to a single \( B \) subobject within the object of type \( AA \), and every other occurrence of a (non-virtual) base class \( B \) in the class lattice of \( AA \) corresponds one-to-one with a distinct \( B \) subobject within the object of type \( AA \). Given the class \( AA \) defined above, class \( AA \) has two subobjects of class \( B \): \( Z \)'s \( B \) and the virtual \( B \) shared by \( X \) and \( Y \), as shown in Figure 5.

---

![Figure 5: Virtual and non-virtual base](fig:class.virtnonvirt)

---

### 11.7.3 Virtual functions

A non-static member function is a virtual function if it is first declared with the keyword `virtual` or if it overrides a virtual member function declared in a base class (see below).

**Note 1**: Virtual functions support dynamic binding and object-oriented programming.

---

A class that declares or inherits a virtual function is called a polymorphic class.

---

111) The use of the `virtual` specifier in the declaration of an overriding function is valid but redundant (has empty semantics).

112) If all virtual functions are immediate functions, the class is still polymorphic even though its internal representation might not otherwise require any additions for that polymorphic behavior.
2 If a virtual member function `vf` is declared in a class `Base` and in a class `Derived`, derived directly or indirectly from `Base`, a member function `vf` with the same name, parameter-type-list (9.3.4.6), cv-qualification, and ref-qualifier (or absence of same) as `Base::vf` is declared, then `Derived::vf` overrides `Base::vf`. For convenience we say that any virtual function overrides itself. A virtual member function `C::vf` of a class object `S` is a final overrider unless the most derived class (6.7.2) of which `S` is a base class subobject (if any) declares or inherits another member function that overrides `vf`. In a derived class, if a virtual member function of a base class subobject has more than one final overrider the program is ill-formed.

[Example 1:]

```cpp
struct A {
    virtual void f();
};
struct B : virtual A {
    virtual void f();
};
struct C : B, virtual A {
    using A::f;
};

void foo() {
    C c;
    c.f();       // calls B::f, the final overrider
    c.C::f();    // calls A::f because of the using-declaration
}

— end example]
```

[Example 2:]

```cpp
struct A { virtual void f(); }
struct B : A { }
struct C : A { void f(); }
struct D : B, C { }       // OK: A::f and C::f are the final overriders
                          // for the B and C subobjects, respectively
```

— end example]

3 [Note 2: A virtual member function does not have to be visible to be overridden, for example,

```cpp
struct B {
    virtual void f();
};
struct D : B {
    void f(int);
};
struct D2 : D {
    void f();
};
```

the function `f(int)` in class `D` hides the virtual function `f()` in its base class `B; D::f(int)` is not a virtual function. However, `f()` declared in class `D2` has the same name and the same parameter list as `B::f()`, and therefore is a virtual function that overrides the function `B::f()` even though `B::f()` is not visible in class `D2`. — end note]

4 If a virtual function `f` in some class `B` is marked with the `virt-specifier final` and in a class `D` derived from `B` a function `D::f` overrides `B::f`, the program is ill-formed.

[Example 3:]

```cpp
struct B {
    virtual void f() const final;
};

struct D : B {
    void f() const;    // error: D::f attempts to override final B::f
};

— end example]

113) A function with the same name but a different parameter list (Clause 12) as a virtual function is not necessarily virtual and does not override. Access control (11.9) is not considered in determining overriding.
If a virtual function is marked with the `virt-specifier override` and does not override a member function of a base class, the program is ill-formed.

[Example 4:

```cpp
struct B {
    virtual void f(int);
};

struct D : B {
    virtual void f(long) override; // error: wrong signature overriding B::f
    virtual void f(int) override; // OK
};
```

— end example]

A virtual function shall not have a trailing `requires-clause` (9.3).

[Example 5:

```cpp
template<typename T>
struct A {
    virtual void f() requires true; // error: virtual function cannot be constrained (13.5.3)
};
```

— end example]

Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see 11.4.7 and 11.12.

The return type of an overriding function shall be either identical to the return type of the overridden function or `covariant` with the classes of the functions. If a function `D::f` overrides a function `B::f`, the return types of the functions are covariant if they satisfy the following criteria:

1. Both are pointers to classes, both are lvalue references to classes, or both are rvalue references to classes.
2. The class in the return type of `B::f` is the same class as the class in the return type of `D::f`, or is an unambiguous and accessible direct or indirect base class of the class in the return type of `D::f`.
3. Both pointers or references have the same cv-qualification and the class type in the return type of `D::f` has the same cv-qualification as or less cv-qualification than the class type in the return type of `B::f`.

If the class type in the covariant return type of `D::f` differs from that of `B::f`, the class type in the return type of `D::f` shall be complete at the point of declaration of `D::f` or shall be the class type `D`. When the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function (7.6.1.3).

[Example 6:

```cpp
class B { }

class D : private B { friend class Derived; }

struct Base {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    virtual B* vf4();
    virtual B* vf5();
    void f();
};

struct No_good : public Base {
    D* vf4(); // error: B (base class of D) inaccessible
};

class A;

struct Derived : public Base {
    void vf1(); // virtual and overrides Base::vf1()
    void vf2(int); // not virtual, hides Base::vf2()
    char vf3(); // error: invalid difference in return type only
}
```

114) Multi-level pointers to classes or references to multi-level pointers to classes are not allowed.
D* vf4(); // OK: returns pointer to derived class
A* vf5(); // error: returns pointer to incomplete class
void f();

};

void g() {
  Derived d;
  Base* bp = &d; // standard conversion:
  // Derived* to Base*
  bp->vf1(); // calls Derived::vf1()
  bp->vf2(); // calls Base::vf2()
  bp->f(); // calls Base::f() (not virtual)
  B* p = bp->vf4(); // calls Derived::vf4() and converts the
  // result to B*
  Derived* dp = &d;
  D* q = dp->vf4(); // calls Derived::vf4() and does not
  // convert the result to B*
  dp->vf2(); // error: argument mismatch
}

—end example]

10 [Note 3: The interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), whereas the interpretation of a call of a non-virtual member function depends only on the type of the pointer or reference denoting that object (the static type) (7.6.1.3). — end note]

11 [Note 4: The virtual specifier implies membership, so a virtual function cannot be a non-member (9.2.3) function. Nor can a virtual function be a static member, since a virtual function call relies on a specific object for determining which function to invoke. A virtual function declared in one class can be declared a friend (11.9.4) in another class. — end note]

12 A virtual function declared in a class shall be defined, or declared pure (11.7.4) in that class, or both; no diagnostic is required (6.3).

13 [Example 7: Here are some uses of virtual functions with multiple base classes:

```cpp
struct A {
  virtual void f();
};

struct B1 : A { // note non-virtual derivation
  void f();
};

struct B2 : A {
  void f();
};

struct D : B1, B2 { // D has two separate A subobjects
};

void foo() {
  D d;
  // A* ap = &d; // would be ill-formed: ambiguous
  B1* b1p = &d;
  A* ap = b1p;
  D* dp = &d;
  ap->f(); // calls D::B1::f
  dp->f(); // error: ambiguous
}
```

In class D above there are two occurrences of class A and hence two occurrences of the virtual member function A::f. The final overrider of B1::A::f is B1::f and the final overrider of B2::A::f is B2::f. — end example]

14 [Example 8: The following example shows a function that does not have a unique final overrider:

```cpp
struct A {
  virtual void f();
};
```
struct VB1 : virtual A { // note virtual derivation
    void f();
};

struct VB2 : virtual A {
    void f();
};

struct Error : VB1, VB2 { // error
};

struct Okay : VB1, VB2 {
    void f();
};

Both VB1::f and VB2::f override A::f but there is no overrider of both of them in class Error. This example is therefore ill-formed. Class Okay is well-formed, however, because Okay::f is a final overrider. —end example

15 [Example 9: The following example uses the well-formed classes from above.

struct VB1a : virtual A { // does not declare f
};

struct Da : VB1a, VB2 {
};

void foe() {
    VB1a* vb1ap = new Da;
    vb1ap->f(); // calls VB2::f
}
—end example]

16 Explicit qualification with the scope operator (7.5.4.3) suppresses the virtual call mechanism.

[Example 10:

    class B { public: virtual void f();
    ;
    class D : public B { public: void f();
    ;
    void D::f() { /* ... */ B::f(); }

Here, the function call in D::f really does call B::f and not D::f. —end example]

17 A function with a deleted definition (9.5) shall not override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition shall not override a function with a deleted definition.

18 A constexpr virtual function shall not override a virtual function that is not constexpr. A constexpr virtual function shall not be overridden by a virtual function that is not constexpr.

11.7.4 Abstract classes [class.abstract]

[Note 1: The abstract class mechanism supports the notion of a general concept, such as a shape, of which only more concrete variants, such as circle and square, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations. —end note]

A virtual function is specified as a pure virtual function by using a pure-specifier (11.4) in the function declaration in the class definition. [Note 2: Such a function might be inherited: see below. —end note]

A class is an abstract class if it has at least one pure virtual function. [Note 3: An abstract class can be used only as a base class of some other class; no objects of an abstract class can be created except as subobjects of a class derived from it (6.2, 11.4). —end note]

A pure virtual function need be defined only if called with, or as if with (11.4.7), the qualified-id syntax (7.5.4.3). [Example 1:

    class point { /* ... */
    ;
    class shape {
        // abstract class
        point center;
    }
public:
  point where() { return center; }
  void move(point p) { center = p; draw(); }
  virtual void rotate(int) = 0; // pure virtual
  virtual void draw() = 0; // pure virtual
};

—end example

[Note 4: A function declaration cannot provide both a pure-specifier and a definition. — end note]

[Example 2:

  struct C {
    virtual void f() = 0 { }; // error
  };

  —end example]

3 [Note 5: An abstract class type cannot be used as a parameter or return type of a function being defined (9.3.4.6) or called (7.6.1.3), except as specified in 9.2.9.3. Further, an abstract class type cannot be used as the type of an explicit type conversion (7.6.1.9, 7.6.1.10, 7.6.1.11), because the resulting prvalue would be of abstract class type (7.2.1). However, pointers and references to abstract class types can appear in such contexts. — end note]

4 A class is abstract if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual.

[Example 3:

  class ab_circle : public shape {
    int radius;
    public:
      void rotate(int) { }
      // ab_circle::draw() is a pure virtual
  };

  Since shape::draw() is a pure virtual function ab_circle::draw() is a pure virtual by default. The alternative declaration,

    class circle : public shape {
      int radius;
      public:
        void rotate(int) { }
        void draw(); // a definition is required somewhere
    };

  would make class circle non-abstract and a definition of circle::draw() must be provided. — end example]

5 [Note 6: An abstract class can be derived from a class that is not abstract, and a pure virtual function can override a virtual function which is not pure. — end note]

6 Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (11.7.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.

11.8 Member name lookup

1 Member name lookup determines the meaning of a name (id-expression) in a class scope (6.4.7). Name lookup can result in an ambiguity, in which case the program is ill-formed. For an unqualified-id, name lookup begins in the class scope of this; for a qualified-id, name lookup begins in the scope of the nested-name-specifier. Name lookup takes place before access control (6.5, 11.9).

2 The following steps define the result of name lookup for a member name f in a class scope C.

3 The lookup set for f in C, called S(f, C), consists of two component sets: the declaration set, a set of members named f; and the subobject set, a set of subobjects where declarations of these members (possibly including using-declarations) were found. In the declaration set, using-declarations are replaced by the set of designated members that are not hidden or overridden by members of the derived class (9.9), and type declarations (including injected-class-names) are replaced by the types they designate. S(f, C) is calculated as follows:

4 If C contains a declaration of the name f, the declaration set contains every declaration of f declared in C that satisfies the requirements of the language construct in which the lookup occurs.
If the resulting declaration set is not empty, the subobject set contains C itself, and calculation is complete. Otherwise (i.e., C does not contain a declaration of f or the resulting declaration set is empty), S(f,C) is initially empty. If C has base classes, calculate the lookup set for f in each direct base class subobject B_i, and merge each such lookup set S(f,B_i) in turn into S(f,C).

The following steps define the result of merging lookup set S(f,B_i) into the intermediate S(f,C):

1. If each of the subobject members of S(f,B_i) is a base class subobject of at least one of the subobject members of S(f,C), or if S(f,B_i) is empty, S(f,C) is unchanged and the merge is complete. Conversely, if each of the subobject members of S(f,C) is a base class subobject of at least one of the subobject members of S(f,B_i), or if S(f,C) is empty, the new S(f,C) is a copy of S(f,B_i).

2. Otherwise, if the declaration sets of S(f,B_i) and S(f,C) differ, the merge is ambiguous: the new S(f,C) is a lookup set with an invalid declaration set and the union of the subobject sets. In subsequent merges, an invalid declaration set is considered different from any other.

3. Otherwise, the new S(f,C) is a lookup set with the shared set of declarations and the union of the subobject sets.

The result of name lookup for f in C is the declaration set of S(f,C). If it is an invalid set, the program is ill-formed.

Example 1:

```c
struct A { int x; }; // S(x,A) = { { A::x }, { A } }
struct B { float x; }; // S(x,B) = { { B::x }, { B } }
struct C: public A, public B { }; // S(x,C) = { invalid, { A in C, B in C } }
struct D: public virtual C { }; // S(x,D) = S(x,C)
struct E: public virtual C { char x; }; // S(x,E) = { { E::x }, { E } }
struct F: public D, public E { }; // S(x,F) = S(x,E)
int main() {
    F f;
    f.x = 0; // OK, lookup finds E::x
}
```

S(x,F) is unambiguous because the A and B base class subobjects of D are also base class subobjects of E, so S(x,D) is discarded in the first merge step. —end example

8 If the name of an overloaded function is unambiguously found, overload resolution (12.4) also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name.

Example 2:

```c
struct A {
    int f();
};
struct B {
    int f();
};
struct C : A, B {
    int f() { return A::f() + B::f(); }
};
—end example
```

9 [Note 2: A static member, a nested type or an enumerator defined in a base class T can unambiguously be found even if an object has more than one base class subobject of type T. Two base class subobjects share the non-static member subobjects of their common virtual base classes. —end note]

Example 3:

```c
struct V {
    int v;
};
```
struct A {
  int a;
  static int s;
  enum { e };
};
struct B : A, virtual V { }
struct C : A, virtual V { }
struct D : B, C { }

void f(D* pd) {
  pd->v++;
  // OK: only one v (virtual)
  pd->s++;
  // OK: only one s (static)
  int i = pd->e;
  // OK: only one e (enumerator)
  pd->a++;
  // error: ambiguous: two a as in D
}

—end example]  

[Note 3: When virtual base classes are used, a hidden declaration can be reached along a path through the subobject lattice that does not pass through the hiding declaration. This is not an ambiguity. The identical use with non-virtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. —end note]

[Example 4:]
struct V { int f(); int x; };
struct W { int g(); int y; };
struct B : virtual V, W {
  int f(); int x;
  int g(); int y;
};
struct C : virtual V, W { }
struct D : B, C { void glorp(); }

struct D : B, C { void glorp(); }

Figure 6: Name lookup [fig:class.lookup]

As illustrated in Figure 6, the names declared in V and the left-hand instance of W are hidden by those in B, but the names declared in the right-hand instance of W are not hidden at all.

void D::glorp() {
  x++;
  // OK: B::x hides V::x
  f();
  // OK: B::f() hides V::f()
  y++;
  // error: B::y and C's W::y
  g();
  // error: B::g() and C's W::g()
}

—end example]

An explicit or implicit conversion from a pointer to or an expression designating an object of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class.

[Example 5:]
struct V { };
struct A { };
struct B : A, virtual V { }
struct C : A, virtual V { };

§ 11.8
struct D : B, C { };

void g() {
    D d;
    B* pb = &d;
    A* pa = &d; // error: ambiguous: C’s A or B’s A?
    V* pv = &d; // OK: only one V subobject
}

—end example

[Note 4: Even if the result of name lookup is unambiguous, use of a name found in multiple subobjects might still be ambiguous (7.3.13, 7.6.1.5, 11.9.3). — end note]

Example 6:

struct B1 {
    void f();
    static void f(int);
    int i;
};
struct B2 {
    void f(double);
};
struct I1: B1 { };
struct I2: B1 { };

struct D: I1, I2, B2 {
    using B1::f;
    using B2::f;
    void g() {
        f(); // Ambiguous conversion of this
        f(0); // Unambiguous (static)
        f(0.0); // Unambiguous (only one B2)
        int B1::* mpB1 = &D::i; // Unambiguous
        int D::* mpD = &D::i; // Ambiguous conversion
    }
};
—end example]

11.9 Member access control [class.access]

11.9.1 General [class.access.general]

1 A member of a class can be

(1.1) — private; that is, its name can be used only by members and friends of the class in which it is declared.

(1.2) — protected; that is, its name can be used only by members and friends of the class in which it is declared, by classes derived from that class, and by their friends (see 11.9.5).

(1.3) — public; that is, its name can be used anywhere without access restriction.

2 A member of a class can also access all the names to which the class has access. A local class of a member function may access the same names that the member function itself may access. 115

3 Members of a class defined with the keyword class are private by default. Members of a class defined with the keywords struct or union are public by default.

[Example 1:

class X {
    int a;
    // X::a is private by default
};

struct S {
    int a;
    // S::a is public by default
};

115) Access permissions are thus transitive and cumulative to nested and local classes.
4 Access control is applied uniformly to all names, whether the names are referred to from declarations or expressions.

[Note 1: Access control applies to names nominated by friend declarations (11.9.4) and using-declarations (9.9). — end note]

In the case of overloaded function names, access control is applied to the function selected by overload resolution.

[Note 2: Because access control applies to names, if access control is applied to a typedef name, only the accessibility of the typedef name itself is considered. The accessibility of the entity referred to by the typedef is not considered. For example,

```cpp
class A {
    class B { };
    public:
        typedef B BB;
};

void f() {
    A::BB x; // OK, typedef name A::BB is public
    A::B y; // access error, A::B is private
}
```

— end note]

5 [Note 3: Access to members and base classes is controlled, not their visibility (6.4.10). Names of members are still visible, and implicit conversions to base classes are still considered, when those members and base classes are inaccessible. — end note]

The interpretation of a given construct is established without regard to access control. If the interpretation established makes use of inaccessible member names or base classes, the construct is ill-formed.

6 All access controls in 11.9 affect the ability to access a class member name from the declaration of a particular entity, including parts of the declaration preceding the name of the entity being declared and, if the entity is a class, the definitions of members of the class appearing outside the class’s member-specification.

[Note 4: This access also applies to implicit references to constructors, conversion functions, and destructors. — end note]

7 [Example 2:

```cpp
class A {
    typedef int I; // private member
    I f();
    friend I g(I);
    static I x;
    template<int> struct Q;
    template<int> friend struct R;
    protected:
        struct B { };
};

A::I A::f() { return 0; }
A::I g(A::I p = A::x);
A::I g(A::I p) { return 0; }
A::I A::x = 0;
template<A::I> struct A::Q { };
template<A::I> struct R { };

struct D: A::B, A { };
```

Here, all the uses of A::I are well-formed because A::f, A::x, and A::Q are members of class A and g and R are friends of class A. This implies, for example, that access checking on the first use of A::I must be deferred until it is determined that this use of A::I is as the return type of a member of class A. Similarly, the use of A::B as a base-specifier is well-formed because D is derived from A, so checking of base-specifiers must be deferred until the entire base-specifier-list has been seen. — end example]
The names in a default argument (9.3.4.7) are bound at the point of declaration, and access is checked at that point rather than at any points of use of the default argument. Access checking for default arguments in function templates and in member functions of class templates is performed as described in 13.9.2.

The names in a default template-argument (13.2) have their access checked in the context in which they appear rather than at any points of use of the default template-argument.

**Example 3:**
```cpp
class B { };  
template <class T> class C {  
    protected:  
        typedef T TT;  
    };  

template <class U, class V = typename U::TT> 
class D : public U { };  

D <C<B> >> d;  // access error, C::TT is protected
```

---

### 11.9.2 Access specifiers

Member declarations can be labeled by an access-specifier (11.7):

```
access-specifier : member-specification_opt
```

An access-specifier specifies the access rules for members following it until the end of the class or until another access-specifier is encountered.

**Example 1:**
```cpp
class X {  
    int a;  // X::a is private by default: class used  
    public:  
        int b;  // X::b is public  
        int c;  // X::c is public  
};  
```

---

Any number of access specifiers is allowed and no particular order is required.

**Example 2:**
```cpp
struct S {  
    int a;  // S::a is public by default: struct used  
    protected:  
        int b;  // S::b is protected  
    private:  
        int c;  // S::c is private  
    public:  
        int d;  // S::d is public  
};  
```

---

**Note 1:** The effect of access control on the order of allocation of data members is specified in 7.6.9. — end note

When a member is redeclared within its class definition, the access specified at its redeclaration shall be the same as at its initial declaration.

**Example 3:**
```cpp
struct S {  
    class A;  
    enum E : int;  
    private:  
        class A {};  // error: cannot change access  
        enum E: int { e0 };  // error: cannot change access  
};  
```

---

§ 11.9.2
[Note 2: In a derived class, the lookup of a base class name will find the injected-class-name instead of the name of the base class in the scope in which it was declared. The injected-class-name might be less accessible than the name of the base class in the scope in which it was declared. —end note]

[Example 4:

```cpp
class A { }
class B : private A { }
class C : public B {
    A* p;  // error: injected-class-name A is inaccessible
    ::A* q;  // OK
};
```

—end example]

### 11.9.3 Accessibility of base classes and base class members  
[Class.access.base]

If a class is declared to be a base class (11.7) for another class using the `public` access specifier, the public members of the base class are accessible as public members of the derived class and protected members of the base class are accessible as protected members of the derived class. If a class is declared to be a base class for another class using the `protected` access specifier, the public and protected members of the base class are accessible as protected members of the derived class. If a class is declared to be a base class for another class using the `private` access specifier, the public and protected members of the base class are accessible as private members of the derived class.\(^{116}\)

In the absence of an `access-specifier` for a base class, `public` is assumed when the derived class is defined with the `class-key struct` and `private` is assumed when the class is defined with the `class-key class`.

[Example 1:

```cpp
class B { /* ... */);
class D1 : private B { /* ... */ );
class D2 : public B { /* ... */ );
class D3 : B { /* ... */ );  // B private by default
struct D4 : public B { /* ... */ );
struct D5 : private B { /* ... */ );
struct D6 : B { /* ... */ );  // B public by default
class D7 : protected B { /* ... */ );
struct D8 : protected B { /* ... */ );
```

Here B is a public base of D2, D4, and D6, a private base of D1, D3, and D5, and a protected base of D7 and D8. —end example]

[Note 1: A member of a private base class might be inaccessible as an inherited member name, but accessible directly. Because of the rules on pointer conversions (7.3.12) and explicit casts (7.6.1.4, 7.6.1.9, 7.6.3), a conversion from a pointer to a derived class to a pointer to an inaccessible base class might be ill-formed if an implicit conversion is used, but well-formed if an explicit cast is used. For example,

```cpp
class B {
    public:
        int mi;  // non-static member
    static int si;  // static member
};
class D : private B {
};
class DD : public D {
    void f();
};

void DD::f() {
    mi = 3;  // error: mi is private in D
    si = 3;  // error: si is private in D
    ::B b;
    b.mi = 3;  // OK (b.mi is different from this->mi)
    b.si = 3;  // OK (b.si is different from this->si)
    ::B::si = 3;  // OK
```

116) As specified previously in 11.9, private members of a base class remain inaccessible even to derived classes unless friend declarations within the base class definition are used to grant access explicitly.
A base class $B$ of $N$ is accessible at $R$, if

1. $R$ occurs in a member or friend of class $N$, and an invented public member of $B$ would be a private or protected member of $N$, or
2. $R$ occurs in a member or friend of a class $P$ derived from $N$, and an invented public member of $B$ would be a private or protected member of $P$, or
3. $R$ occurs in a member or friend of class $N$, and an invented public member of $B$ would be a private or protected member of $N$, or
4. there exists a class $S$ such that $B$ is a base class of $S$ accessible at $R$ and $S$ is a base class of $N$ accessible at $R$.

**Example 2:**

```cpp
class B {
public:
  int m;
};
class S: private B {
  friend class N;
};
class N: private S {
  void f() {
    B* p = this; // OK because class S satisfies the fourth condition above: B is a base class of N
    // accessible in f() because B is an accessible base class of S and S is an accessible // base class of N.
  }
};
```

**Example 3:**

```cpp
class B;
class A {
  private:
    int i;
    friend void f(B*);
};
```
class B : public A { };  
void f(B* p) {  
    p->i = 1;  // OK: B* can be implicitly converted to A*, and f has access to i in A  
}  
— end example

If a class member access operator, including an implicit “this->”, is used to access a non-static data member or non-static member function, the reference is ill-formed if the left operand (considered as a pointer in the “.” operator case) cannot be implicitly converted to a pointer to the naming class of the right operand.

[Note 4: This requirement is in addition to the requirement that the member be accessible as named. — end note]

11.9.4 Friends

A friend of a class is a function or class that is given permission to use the private and protected member names from the class. A class specifies its friends, if any, by way of friend declarations. Such declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class.

[Example 1: The following example illustrates the differences between members and friends:

class X {
    int a;
    friend void friend_set(X*, int);
public:
    void member_set(int);
};

void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }

void f() {
    X obj;
    friend_set(&obj,10);
    obj.member_set(10);
}
— end example]

Declaring a class to be a friend implies that the names of private and protected members from the class granting friendship can be accessed in the base-specifiers and member declarations of the befriended class.

[Example 2:

class A {
    class B { };  
    friend class X;
};

struct X : A::B {  // OK: A::B accessible to friend
    A::B mx;
    // OK: A::B accessible to member of friend
    class Y {
        A::B my;
        // OK: A::B accessible to nested member of friend
    };
};  
— end example]

[Example 3:

class X {
    enum { a=100 }
    friend class Y;
};

class Y {
    int v[X::a];  // OK, Y is a friend of X
};
A class shall not be defined in a friend declaration.

Example 4:

```cpp
class A {
    friend class B { };  // error: cannot define class in friend declaration
};
```

Example 5:

```cpp
class C;
typedef C Ct;

class X1 {
    friend C;  // OK: class C is a friend
};

class X2 {
    friend Ct;
    friend D;   // error: no type-name D in scope
    friend class D;  // OK: elaborated-type-specifier declares new class
};
template <typename T> class R {
    friend T;
};
R<C> rc;  // class C is a friend of R<C>
R<int> Ri;  // OK: "friend int;" is ignored
```

Example 6:

```cpp
class Y {
    friend char* X::foo(int);
    friend X::*X(char);  // constructors can be friends
    friend X::*-X();  // destructors can be friends
};
```

Example 7:

```cpp
class M {
    friend void f() { }  // definition of global f, a friend of M,
};
```
Such a function is implicitly an inline (9.2.8) function if it is attached to the global module. A friend function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not (6.5.2).

No storage-class-specifier shall appear in the decl-specifier-seq of a friend declaration.

A name nominated by a friend declaration shall be accessible in the scope of the class containing the friend declaration. The meaning of the friend declaration is the same whether the friend declaration appears in the private, protected, or public (11.4) portion of the class member-specification.

Friendship is neither inherited nor transitive.

Example 8:
```cpp
class A {
  friend class B;
  int a;
};

class B {
  friend class C;
};

class C {
  void f(A* p) { // error: C is not a friend of A despite being a friend of a friend
    p->a++;
  }
};

class D : public B {
  void f(A* p) { // error: D is not a friend of A despite being derived from a friend
    p->a++;
  }
};
```

Example 9:
```cpp
class X;
void a();
void f() {
  class Y;
  extern void b();
  class A {
    friend class X; // OK, but X is a local class, not ::X
    friend class Y; // OK
    friend class Z; // OK, introduces local class Z
    friend void a(); // error, ::a is not considered
    friend void b(); // OK
    friend void c(); // error
  };
  X* px; // OK, but ::X is found
  Z* pz; // error: no Z is found
}
```

If a friend declaration appears in a local class (11.6) and the name specified is an unqualified name, a prior declaration is looked up without considering scopes that are outside the innermost enclosing non-class scope.

For a friend function declaration, if there is no prior declaration, the program is ill-formed. For a friend class declaration, if there is no prior declaration, the class that is specified belongs to the innermost enclosing non-class scope, but if it is subsequently referenced, its name is not found by name lookup until a matching declaration is provided in the innermost enclosing non-class scope.

Example 9:
11.9.5 Protected member access

An additional access check beyond those described earlier in 11.9 is applied when a non-static data member or non-static member function is a protected member of its naming class (11.9.3). As described earlier, access to a protected member is granted because the reference occurs in a friend or member of some class C. If the access is to form a pointer to member (7.6.2.2), the nested-name-specifier shall denote C or a class derived from C. All other accesses involve a (possibly implicit) object expression (7.6.1.5). In this case, the class of the object expression shall be C or a class derived from C.

[Example 1:

```cpp
class B {
  protected:
    int i;
    static int j;
};

class D1 : public B {
};

class D2 : public B {
  friend void fr(B*, D1*, D2*);
  void mem(B*, D1*);
};

void fr(B* pb, D1* p1, D2* p2) {
  pb->i = 1;          // error
  p1->i = 2;          // error
  p2->i = 3;          // OK (access through a D2)
  p2->B::i = 4;       // OK (access through B2, even though naming class is B)
  int B::* pmi_B = &B::i; // error
  int B::* pmi_B2 = &D2::i; // OK (type of &D2::i is int B::*
  B::j = 5;          // error: not a friend of naming class B
  D2::j = 6;         // OK (because refers to static member)
}

void D2::mem(B* pb, D1* p1) {
  pb->i = 1;          // error
  p1->i = 2;          // error
  i = 3;              // OK (access through this)
  B::i = 4;           // OK (access through this, qualification ignored)
  int B::* pmi_B = &B::i; // error
  int B::* pmi_B2 = &D2::i; // OK
  j = 5;              // OK (because j refers to static member)
  B::j = 6;           // OK (because B::j refers to static member)
}

void g(B* pb, D1* p1, D2* p2) {
  pb->i = 1;          // error
  p1->i = 2;          // error
  p2->i = 3;          // error
}
@end example]

11.9.6 Access to virtual functions

The access rules (11.9) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it.

[Example 1:

```cpp
class B {
  public:
    virtual int f();
};
```
class D : public B {
    private:
    int f();
};

void f() {
    D d;
    B* pb = &d;
    D* pd = &d;
    // OK: B::f() is public, D::f() is invoked
    pb->f();
    // error: D::f() is private
    pd->f();
}

—end example]

Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (B* in the example above). The access of the member function in the class in which it was defined (D in the example above) is in general not known.

11.9.7 Multiple access [class.paths]
1 If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access.
[Example 1:
    class W { public: void f(); };  
    class A : private virtual W { };  
    class B : public virtual W { }; 
    class C : public A, public B {
        void f() { W::f(); }          // OK
    }
]
Since W::f() is available to C::f() along the public path through B, access is allowed. —end example]

11.9.8 Nested classes [class.access.nest]
1 A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules (11.9) shall be obeyed.
[Example 1:
    class E {
        int x;
        class B { };  
        class I {
            B b;              // OK: E::I can access E::B
            int y;
            void f(E* p, int i) {
                p->x = i;        // OK: E::I can access E::x
            }
        }
        int g(I* p) {
            return p->y;     // error: I::y is private
        }
    }
—end example]

11.10 Initialization [class.init]

11.10.1 General [class.init.general]
1 When no initializer is specified for an object of (possibly cv-qualified) class type (or array thereof), or the initializer has the form (), the object is initialized as specified in 9.4.
2 An object of class type (or array thereof) can be explicitly initialized; see 11.10.2 and 11.10.3.
When an array of class objects is initialized (either explicitly or implicitly) and the elements are initialized by constructor, the constructor shall be called for each element of the array, following the subscript order; see 9.3.4.5.

[Note 1: Destructors for the array elements are called in reverse order of their construction. — end note]

11.10.2 Explicit initialization [class.expl.init]

An object of class type can be initialized with a parenthesized expression-list, where the expression-list is construed as an argument list for a constructor that is called to initialize the object. Alternatively, a single assignment-expression can be specified as an initializer using the = form of initialization. Either direct-initialization semantics or copy-initialization semantics apply; see 9.4.

[Example 1:

```c
struct complex {
    complex();
    complex(double);
    complex(double,double);
};
complex sqrt(complex,complex);

complex a(1);       // initialized by calling complex(double) with argument 1
complex b = a;      // initialized as a copy of a
complex c = complex(1,2); // initialized by calling complex(double,double) with arguments 1 and 2
complex d = sqrt(b,c); // initialized by calling sqrt(complex,complex) with d as its result object
complex e;
complex f = 3;      // initialized by calling complex(double) with argument 3
complex g = {1, 2}; // initialized by calling complex(double, double) with arguments 1 and 2
```
—end example]

[Note 1: Overloading of the assignment operator (12.6.3.2) has no effect on initialization. — end note]

2 An object of class type can also be initialized by a braced-init-list. List-initialization semantics apply; see 9.4 and 9.4.5.

[Example 2:

```c
complex v[6] = { 1, complex(1,2), complex(), 2 };  
```

Here, complex::complex(double) is called for the initialization of v[0] and v[3], complex::complex(double, double) is called for the initialization of v[1], complex::complex() is called for the initialization v[2], v[4], and v[5]. For another example,

```c
struct X {
    int i;
    float f;
    complex c;
} x = { 99, 88.8, 77.7 };  
```

Here, x.i is initialized with 99, x.f is initialized with 88.8, and complex::complex(double) is called for the initialization of x.c. — end example]

[Note 2: Braces can be elided in the initializer-list for any aggregate, even if the aggregate has members of a class type with user-defined type conversions; see 9.4.2. — end note]

3 [Note 3: If T is a class type with no default constructor, any declaration of an object of type T (or array thereof) is ill-formed if no initializer is explicitly specified (see 11.10 and 9.4). — end note]

4 [Note 4: The order in which objects with static or thread storage duration are initialized is described in 6.9.3.3 and 8.8. — end note]

11.10.3 Initializing bases and members [class.base.init]

In the definition of a constructor for a class, initializers for direct and virtual base class subobjects and non-static data members can be specified by a ctor-initializer, which has the form

```
cTOR-initializer:
    : mem-initializer-list
```

§ 11.10.3
In a `mem-initializer-id` an initial unqualified identifier is looked up in the scope of the constructor’s class and, if not found in that scope, it is looked up in the scope containing the constructor’s definition. [Note 1: If the constructor’s class contains a member with the same name as a direct or virtual base class of the class, a `mem-initializer-id` naming the member or base class and composed of a single identifier refers to the class member. A `mem-initializer-id` for the hidden base class can be specified using a qualified name. — end note] Unless the `mem-initializer-id` names the constructor’s class, a non-static data member of the constructor’s class, or a direct or virtual base of that class, the `mem-initializer` is ill-formed.

3 A `mem-initializer-list` can initialize a base class using any `class-or-decltype` that denotes that base class type.

[Example 1:

```c
struct A { A(); }
typedef A global_A;
struct B { }
struct C: public A, public B { C(); }
C::C(): global_A() {} // mem-initializer for base A
```
—end example]

4 If a `mem-initializer-id` is ambiguous because it designates both a direct non-virtual base class and an inherited virtual base class, the `mem-initializer` is ill-formed.

[Example 2:

```c
struct A { A(); }
struct B: public virtual A { }
struct C: public A, public B { C(); }
C::C(): A() {} // error: which A?
```
—end example]

5 A `ctor-initializer` may initialize a variant member of the constructor’s class. If a `ctor-initializer` specifies more than one `mem-initializer` for the same member or for the same base class, the `ctor-initializer` is ill-formed.

6 A `mem-initializer-list` can delegate to another constructor of the constructor’s class using any `class-or-decltype` that denotes the constructor’s class itself. If a `mem-initializer-id` designates the constructor’s class, it shall be the only `mem-initializer`; the constructor is a delegating constructor, and the constructor selected by the `mem-initializer` is the target constructor. The target constructor is selected by overload resolution. Once the target constructor returns, the body of the delegating constructor is executed. If a constructor delegates to itself directly or indirectly, the program is ill-formed, no diagnostic required.

[Example 3:

```c
struct C {
    C( int ) {} // #1: non-delegating constructor
    C( int ) {} // #2: delegates to #1
    C( char c ): C(42.0) {} // #3: ill-formed due to recursion with #4
    C( double d ) : C(‘a’) {} // #4: ill-formed due to recursion with #3
};
```
—end example]

7 The `expression-list` or `braced-init-list` in a `mem-initializer` is used to initialize the designated subobject (or, in the case of a delegating constructor, the complete class object) according to the initialization rules of 9.4 for direct-initialization.

[Example 4:

```c
struct B1 { B1(int); /* ... */ }
struct B2 { B2(int); /* ... */ }
```
struct D : B1, B2 {
    D(int);
    B1 b;
    const int c;
};

D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4) { /* ... */ }
D d(10);

— end example

[Note 2: The initialization performed by each mem-initializer constitutes a full-expression (6.9.1). Any expression in a mem-initializer is evaluated as part of the full-expression that performs the initialization. — end note]

A mem-initializer where the mem-initializer-id denotes a virtual base class is ignored during execution of a constructor of any class that is not the most derived class.

8 A temporary expression bound to a reference member in a mem-initializer is ill-formed.

[Example 5:

```c
struct A {
    A() : v(42) { }
    const int& v;
};

— end example
```

9 In a non-delegating constructor, if a given potentially constructed subobject is not designated by a mem-initializer-id (including the case where there is no mem-initializer-list because the constructor has no ctor-initializer), then

(9.1) — if the entity is a non-static data member that has a default member initializer (11.4) and either

(9.1.1) — the constructor’s class is a union (11.5), and no other variant member of that union is designated by a mem-initializer-id or

(9.1.2) — the constructor’s class is not a union, and, if the entity is a member of an anonymous union, no other member of that union is designated by a mem-initializer-id,

the entity is initialized from its default member initializer as specified in 9.4;

(9.2) — otherwise, if the entity is an anonymous union or a variant member (11.5.2), no initialization is performed;

(9.3) — otherwise, the entity is default-initialized (9.4).

[Note 3: An abstract class (11.7.4) is never a most derived class, thus its constructors never initialize virtual base classes, therefore the corresponding mem-initializers can be omitted. — end note]

An attempt to initialize more than one non-static data member of a union renders the program ill-formed.

[Note 4: After the call to a constructor for class X for an object with automatic or dynamic storage duration has completed, if the constructor was not invoked as part of value-initialization and a member of X is neither initialized nor given a value during execution of the compound-statement of the body of the constructor, the member has an indeterminate value. — end note]

[Example 6:

```c
struct A {
    A();
};

struct B {
    B(int);
};

struct C {
    C() { }
    // initializes members as follows:
    A a;
    // OK: calls A::A()
    const B b;
    // error: B has no default constructor
    int i;
    // OK: i has indeterminate value
    int j = 5;
    // OK: j has the value 5
};
```

§ 11.10.3
If a given non-static data member has both a default member initializer and a mem-initializer, the initialization specified by the mem-initializer is performed, and the non-static data member’s default member initializer is ignored.

[Example 7: Given

```cpp
struct A {
    int i = /* some integer expression with side effects */ ;
    A(int arg) : i(arg) { }
        // ...
};
```
]
the A(int) constructor will simply initialize i to the value of arg, and the side effects in i’s default member initializer will not take place. —end example]  

A temporary expression bound to a reference member from a default member initializer is ill-formed.

[Example 8:

```cpp
struct A {
    A() = default; // OK
    A(int v) : v(v) { } // OK
    const int& v = 42; // OK
};
A a1; // error: ill-formed binding of temporary to reference
A a2(1); // OK, unfortunately
```
]  

In a non-delegating constructor, the destructor for each potentially constructed subobject of class type is potentially invoked (11.4.7).

[Note 5: This provision ensures that destructors can be called for fully-constructed subobjects in case an exception is thrown (14.3). —end note]

In a non-delegating constructor, initialization proceeds in the following order:

1. First, and only for the constructor of the most derived class (6.7.2), virtual base classes are initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where “left-to-right” is the order of appearance of the base classes in the derived class `base-specifier-list`.

2. Then, direct base classes are initialized in declaration order as they appear in the `base-specifier-list` (regardless of the order of the mem-initializers).

3. Then, non-static data members are initialized in the order they were declared in the class definition (again regardless of the order of the mem-initializers).

4. Finally, the compound-statement of the constructor body is executed.

[Note 6: The declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. —end note]

[Example 9:

```cpp
struct V {
    V();
    V(int);
};
struct A : virtual V {
    A();
    A(int);
};
struct B : virtual V {
    B();
    B(int);
};
struct C : A, B, virtual V {
    C();
};
```
Names in the expression-list or braced-init-list of a mem-initializer are evaluated in the scope of the constructor for which the mem-initializer is specified.

[Example 10]:

class X {
  int a;
  int b;
  int i;
  int j;
public:
  const int& r;
  X(int i): r(a), b(i), i(i), j(this->i) {
  }
};

initializes X::r to refer to X::a, initializes X::b with the value of the constructor parameter i, initializes X::i with the value of the constructor parameter i, and initializes X::j with the value of X::i; this takes place each time an object of class X is created. —end example]

[Note 7]: Because the mem-initializer are evaluated in the scope of the constructor, the this pointer can be used in the expression-list of a mem-initializer to refer to the object being initialized. —end note]

Member functions (including virtual member functions, 11.7.3) can be called for an object under construction. Similarly, an object under construction can be the operand of the typeid operator (7.6.1.8) or of a dynamic_cast (7.6.1.7). However, if these operations are performed in a ctor-initializer (or in a function called directly or indirectly from a ctor-initializer) before all the mem-initializers for base classes have completed, the program has undefined behavior.

[Example 11]:

class A {
public:
  A(int);
};

class B : public A {
  int j;
public:
  int f();
  B() : A(f()), // undefined behavior: calls member function but base A not yet initialized
    j(f()) { } // well-defined: bases are all initialized
};

class C {
public:
  C(int);
};

class D : public B, C {
  int i;
public:
  D() : C(f()), // undefined behavior: calls member function but base C not yet initialized
    i(f()) { } // well-defined: bases are all initialized
};
A mem-initializer followed by an ellipsis is a pack expansion (13.7.4) that initializes the base classes specified by a pack expansion in the base-specifier-list for the class.

### Example 12:

```
template<class... Mixins>
class X : public Mixins...
{
public:
    X(const Mixins&... mixins) : Mixins(mixins)... { }
};
```

### 11.10.4 Initialization by inherited constructor

When a constructor for type $B$ is invoked to initialize an object of a different type $D$ (that is, when the constructor was inherited (9.9)), initialization proceeds as if a defaulted default constructor were used to initialize the $D$ object and each base class subobject from which the constructor was inherited, except that the $B$ subobject is initialized by the invocation of the inherited constructor. The complete initialization is considered to be a single function call; in particular, the initialization of the inherited constructor’s parameters is sequenced before the initialization of any part of the $D$ object.

### Example 1:

```
struct B1 {
    B1(int, ...) { }
};

struct B2 {
    B2(double) { }
};

int get();

struct D1 : B1 {
    using B1::B1; // inherits B1(int, ...)
    int x;
    int y = get();
};

void test() {
    D1 d(2, 3, 4); // OK: B1 is initialized by calling B1(2, 3, 4),
    // then d.x is default-initialized (no initialization is performed),
    // then d.y is initialized by calling get()
    D1 e; // error: D1 has a deleted default constructor
}

struct D2 : B2 {
    using B2::B2;
    B1 b;
};

D2 f(1.0); // error: B1 has a deleted default constructor
```

### Example 2:

```
struct W { W(int); };
struct X : virtual W { using W::W; X() = delete; };
struct Y : X { using X::X; };
struct Z : Y, virtual W { using Y::Y; };
Z z(0); // OK: initialization of Y does not invoke default constructor of X
```

```
template<class T> struct Log : T {
    using T::T; // inherits all constructors from class T
```
Class template Log wraps any class and forwards all of its constructors, while writing a message to the standard log whenever an object of class Log is destroyed. — end example

If the constructor was inherited from multiple base class subobjects of type B, the program is ill-formed.

Example 2:

```cpp
struct A { A(int); }
struct B : A { using A::A; }

struct C1 : B { using B::B; }
struct C2 : B { using B::B; }

struct D1 : C1, C2 {
    using C1::C1;
    using C2::C2;
}

struct V1 : virtual B { using B::B; }
struct V2 : virtual B { using B::B; }

struct D2 : V1, V2 {
    using V1::V1;
    using V2::V2;
}

D1 d1(0); // error: ambiguous
D2 d2(0); // OK: initializes virtual B base class, which initializes the A base class
// then initializes the V1 and V2 base classes as if by a defaulted default constructor

struct M { M(); M(int); }
struct N : M { using M::M; }
struct O : M {}
struct P : N, O { using N::N; using O::O; }
P p(0); // OK: use M(0) to initialize N's base class,
// use M() to initialize O's base class
// - end example
```

When an object is initialized by an inherited constructor, initialization of the object is complete when the initialization of all subobjects is complete.

11.10.5 Construction and destruction [class.cdtor]

For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior. For an object with a non-trivial destructor, referring to any non-static member or base class of the object after the destructor finishes execution results in undefined behavior.

Example 1:

```cpp
struct X { int i; }
struct Y : X { Y(); }
struct A { int a; }
struct B : public A { int j; Y y; }

extern B bobj;
B* pb = &bobj; // OK
int* p1 = &bobj.a; // undefined behavior: refers to base class member
int* p2 = &bobj.y.i; // undefined behavior: refers to member's member

A* pa = &bobj; // undefined behavior: upcast to a base class type
B bobj; // definition of bobj

extern X xobj;
int* p3 = &xobj.i; // OK, X is a trivial class
```
X xobj;

For another example,

```cpp
struct W { int j; 
};
struct X : public virtual W { 
int* p;
X x;
Y() : p(&x.j) { // undefined, x is not yet constructed }
};
```

—end example

2 During the construction of an object, if the value of the object or any of its subobjects is accessed through a glvalue that is not obtained, directly or indirectly, from the constructor’s this pointer, the value of the object or subobject thus obtained is unspecified.

[Example 2]:

```cpp
struct C;
void no_opt(C*);

struct C {
    int c;
    C() : c(0) { no_opt(this); } 
};

const C cobj;

void no_opt(C* cptr) {
    int i = cobj.c * 100; // value of cobj.c is unspecified
    cptr->c = 1;
    cout << cobj.c * 100 // value of cobj.c is unspecified
        << \n; 
}
```

 extern struct D d;
 struct D {
    D(int a) : a(a), b(d.a) {}
    int a, b;
    
};

D d = D(1); // value of d.b is unspecified

—end example

3 To explicitly or implicitly convert a pointer (a glvalue) referring to an object of class X to a pointer (reference) to a direct or indirect base class B of X, the construction of X and the construction of all of its direct or indirect bases that directly or indirectly derive from B shall have started and the destruction of these classes shall not have completed, otherwise the conversion results in undefined behavior. To form a pointer to (or access the value of) a direct non-static member of an object obj, the construction of obj shall have started and its destruction shall not have completed, otherwise the computation of the pointer value (or accessing the member value) results in undefined behavior.

[Example 3]:

```cpp
struct A { 
};
struct B : virtual A { 
};
struct C : B { 
};
struct D : virtual A { D(A*); 
};
struct X { X(A*); 
};
```

```cpp
struct E : C, D, X {
    E() : D(this), // undefined behavior: upcast from E* to A* might use path E* -> D* -> A*
        // but D is not constructed

    // “D((C*)this)” would be defined: E* -> C* is defined because E() has started,
    // and C* -> A* is defined because C is fully constructed
```
Member functions, including virtual functions (11.7.3), can be called during construction or destruction (11.10.3). When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, and the object to which the call applies is the object (call it x) under construction or destruction, the function called is the final overrider in the constructor’s or destructor’s class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access (7.6.1.5) and the object expression refers to the complete object of x or one of that object’s base class subobjects but not x or one of its base class subobjects, the behavior is undefined.

Example 4:

```cpp
struct V {
    virtual void f();
    virtual void g();
};

struct A : virtual V {
    virtual void f();
};

struct B : virtual V {
    virtual void g();
    B(V*, A*);
};

struct D : A, B {
    virtual void f();
    virtual void g();
    D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
    f(); // calls V::f, not A::f
    g(); // calls B::g, not D::g
    v->g(); // v is base of B, the call is well-defined, calls B::g
    a->f(); // undefined behavior: a’s type not a base of B
}
```

The `typeid` operator (7.6.1.8) can be used during construction or destruction (11.10.3). When `typeid` is used in a constructor (including the mem-initializer or default member initializer (11.4) for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of `typeid` refers to the object under construction or destruction, `typeid` yields the `std::type_info` object representing the constructor or destructor’s class. If the operand of `typeid` refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor’s class nor one of its bases, the behavior is undefined.

Dynamic casts (7.6.1.7) can be used during construction or destruction (11.10.3). When a `dynamic_cast` is used in a constructor (including the mem-initializer or default member initializer for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of the `dynamic_cast` refers to the object under construction or destruction, this object is considered to be a most derived object that has the type of the constructor or destructor’s class. If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor’s own class or one of its bases, the `dynamic_cast` results in undefined behavior.
struct A : virtual V { }

struct B : virtual V {
    B(V*, A*);
};

struct D : A, B {
    D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
    typeid(*this);
    // well-defined: *v has type V, a base of B yields type_info for B
    typeid(*v);
    // well-defined: *v of type V*, V base of B results in B*  
    dynamic_cast<B*>(v);
    // undefined behavior: a has type A*, A not a base of B
    dynamic_cast<B*>(a);
}

11.10.6 Copy/move elision

When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object. If the first parameter of the selected constructor is an rvalue reference to the object’s type, the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization. This elision of copy/move operations, called copy elision, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

1. In a return statement in a function (14.8) with a class return type, when the expression is the name of a non-volatile object with automatic storage duration (other than a function parameter or a variable introduced by the exception-declaration of a handler (14.4)) with the same type (ignoring cv-qualification) as the function return type, the copy/move operation can be omitted by constructing the object directly into the function call’s return object.

2. In a throw-expression (7.6.18), when the operand is the name of a non-volatile object with automatic storage duration (other than a function or catch-clause parameter) whose scope does not extend beyond the end of the innermost enclosing try-block (if there is one), the copy/move operation can be omitted by constructing the object directly into the exception object.

3. In a coroutine (9.5.4), a copy of a coroutine parameter can be omitted and references to that copy replaced with references to the corresponding parameter if the meaning of the program will be unchanged except for the execution of a constructor and destructor for the parameter copy object.

4. When the exception-declaration of an exception handler (14.1) declares an object of the same type (except for cv-qualification) as the exception object (14.2), the copy operation can be omitted by treating the exception-declaration as an alias for the exception object if the meaning of the program will be unchanged except for the execution of constructors and destructors for the object declared by the exception-declaration.

[Note 1: There cannot be a move from the exception object because it is always an lvalue. — end note]

Copy elision is not permitted where an expression is evaluated in a context requiring a constant expression (7.7) and in constant initialization (6.9.3.2).

[Note 2: Copy elision might be performed if the same expression is evaluated in another context. — end note]

Example 1:

```cpp
class Thing {
    public:
        Thing();
        -Thing();
```
Here the criteria for elision can eliminate the copying of the object \( t \) with automatic storage duration into the result object for the function call \( f() \), which is the global object \( t2 \). Effectively, the construction of the local object \( t \) can be viewed as directly initializing the global object \( t2 \), and that object's destruction will occur at program exit. Adding a move constructor to \( \text{Thing} \) has the same effect, but it is the move construction from the object with automatic storage duration to \( t2 \) that is elided. — end example

An \textit{implicitly movable entity} is a variable of automatic storage duration that is either a non-volatile object or an rvalue reference to a non-volatile object type. In the following copy-initialization contexts, a move operation is first considered before attempting a copy operation:

1. If the \textit{expression} in a \texttt{return} (8.7.4) or \texttt{co_return} (8.7.5) statement is a (possibly parenthesized) \textit{id-expression} that names an implicitly movable entity declared in the body or \texttt{parameter-declaration-clause} of the innermost enclosing function or \texttt{lambda-expression}, or
2. if the operand of a \texttt{throw-expression} (7.6.18) is a (possibly parenthesized) \textit{id-expression} that names an implicitly movable entity whose scope does not extend beyond the \texttt{compound-statement} of the innermost \texttt{try-block} or \texttt{function-try-block} (if any) whose \texttt{compound-statement} or \texttt{ctor-initializer} encloses the \texttt{throw-expression},

overload resolution to select the constructor for the copy or the \texttt{return_value} overload to call is first performed as if the expression or operand were an rvalue. If the first overload resolution fails or was not performed, overload resolution is performed again, considering the expression or operand as an lvalue.

\textit{Note 3:} This two-stage overload resolution is performed regardless of whether copy elision will occur. It determines the constructor or the \texttt{return_value} overload to be called if elision is not performed, and the selected constructor or \texttt{return_value} overload must be accessible even if the call is elided. — end note

\textbf{Example 2:}

```cpp
class Thing {
public:
  Thing();
  -Thing();
  Thing(Thing&&);
private:
  Thing(const Thing&);
};

Thing f(bool b) {
  Thing t;
```
if (b)
    throw t;   // OK: Thing(Thing&&) used (or elided) to throw t
return t;    // OK: Thing(Thing&&) used (or elided) to return t
}

Thing t2 = f(false);   // OK: no extra copy/move performed, t2 constructed by call to f

struct Weird {
    Weird();
    Weird(Weird&);
};

Weird g() {
    Weird w;
    return w;   // OK: first overload resolution fails, second overload resolution selects Weird(Weird&)
}
—end example

[Example 3:]
template<class T> void g(const T&);

template<class T> void f() {
    T x;
    try {
        T y;
        try { g(x); }
        catch (...) {
            if (/\*...*/)
                throw x;   // does not move
            throw y;    // moves
        }
        g(y);
    } catch(...) {
        g(x);
        g(y);       // error: y is not in scope
    }
}
—end example

11.11 Comparisons [class.compare]

11.11.1 Defaulted comparison operator functions [class.compare.default]

A defaulted comparison operator function (12.6.3) for some class \(C\) shall be a non-template function that is

1. (1.1) a non-static const non-volatile member of \(C\) having one parameter of type const \(C\&) and either no ref-qualifier or the ref-qualifier \&,

1. (1.2) a friend of \(C\) having either two parameters of type const \(C\&) or two parameters of type \(C\).

A comparison operator function for class \(C\) that is defaulted on its first declaration and is not defined as deleted is implicitly defined when it is odr-used or needed for constant evaluation. Name lookups in the defaulted definition of a comparison operator function are performed from a context equivalent to its function-body. A definition of a comparison operator as defaulted that appears in a class shall be the first declaration of that function.

2 A defaulted \(<=\) or \(==\) operator function for class \(C\) is defined as deleted if any non-static data member of \(C\) is of reference type or \(C\) has variant members (11.5.2).

3 A binary operator expression \(a \& b\) is usable if either

3. (3.1) \(a\) or \(b\) is of class or enumeration type and overload resolution (12.4) as applied to \(a \& b\) results in a usable candidate, or

3. (3.2) neither \(a\) nor \(b\) is of class or enumeration type and \(a \& b\) is a valid expression.
A defaulted comparison function is `constexpr-compatible` if it satisfies the requirements for a `constexpr` function (9.2.6) and no overload resolution performed when determining whether to delete the function results in a usable candidate that is a non-`constexpr` function.

[Note 1: This includes the overload resolutions performed:
(4.1) — for an `operator<=>` whose return type is not `auto`, when determining whether a synthesized three-way comparison is defined,
(4.2) — for an `operator<=>` whose return type is `auto` or for an `operator==`, for a comparison between an element of the expanded list of subobjects and itself, or
(4.3) — for a secondary comparison operator @, for the expression x @ y.
— end note]

5 If the `member-specification` does not explicitly declare any member or friend named `operator==`, an `==` operator function is declared implicitly for each three-way comparison operator function defined as defaulted in the `member-specification`, with the same access and `function-definition` and in the same class scope as the respective three-way comparison operator function, except that the return type is replaced with `bool` and the `declarator-id` is replaced with `operator==`.

[Note 2: Such an implicitly-declared `==` operator for a class X is defined as defaulted in the definition of X and has the same `parameter-declaration-clause` and trailing `requires-clause` as the respective three-way comparison operator. It is declared with `friend, virtual, constexpr`, or `consteval` if the three-way comparison operator function is so declared. If the three-way comparison operator function has no `noexcept-specifier`, the implicitly-declared `==` operator function has an implicit exception specification (14.5) that can differ from the implicit exception specification of the three-way comparison operator function. — end note]

[Example 1:
```cpp
template<
    typename T
>
struct X {
  friend constexpr std::partial_ordering operator<=>(X, X) requires (sizeof(T) != 1) = default;
  // implicitly declares: friend constexpr bool operator==(X, X) requires (sizeof(T) != 1) = default;

  [[nodiscard]] virtual std::strong_ordering operator<=>(const X&) const = default;
  // implicitly declares: [[nodiscard]] virtual bool operator==(const X&) const = default;
};
```
— end example]

[Note 3: The `==` operator function is declared implicitly even if the defaulted three-way comparison operator function is defined as deleted. — end note]

6 The direct base class subobjects of C, in the order of their declaration in the `base-specifier-list` of C, followed by the non-static data members of C, in the order of their declaration in the `member-specification` of C, form a list of subobjects. In that list, any subobject of array type is recursively expanded to the sequence of its elements, in the order of increasing subscript. Let \( x_i \) be an lvalue denoting the \( i \)th element in the expanded list of subobjects for an object \( x \) (of length \( n \)), where \( x_i \) is formed by a sequence of derived-to-base conversions (12.4.4.2), class member access expressions (7.6.1.5), and array subscript expressions (7.6.1.2) applied to \( x \).

### 11.11.2 Equality operator

1 A defaulted equality operator function (12.6.3) shall have a declared return type `bool`.

2 A defaulted `==` operator function for a class C is defined as deleted unless, for each \( x_i \) in the expanded list of subobjects for an object \( x \) of type C, \( x_i == x \) is usable (11.11.1).

3 The return value \( V \) of a defaulted `==` operator function with parameters \( x \) and \( y \) is determined by comparing corresponding elements \( x_i \) and \( y_i \) in the expanded lists of subobjects for \( x \) and \( y \) (in increasing index order) until the first index \( i \) where \( x_i == y \) yields a result value which, when contextually converted to `bool`, yields `false`. If no such index exists, \( V \) is `true`. Otherwise, \( V \) is `false`.

[Example 1:
```cpp
struct D {
  int i;
  friend bool operator==(const D& x, const D& y) = default;
  // OK, returns x.i == y.i
};
```
— end example]
11.11.3 Three-way comparison

The synthesized three-way comparison of type $R$ (17.11.2) of glvalues $a$ and $b$ of the same type is defined as follows:

1. If $a <=> b$ is usable (11.11.1), $\text{static\_cast}<R>(a <=> b)$.
2. Otherwise, if overload resolution for $a <=> b$ is performed and finds at least one viable candidate, the synthesized three-way comparison is not defined.
3. Otherwise, if $R$ is not a comparison category type, or either the expression $a == b$ or the expression $a < b$ is not usable, the synthesized three-way comparison is not defined.
4. Otherwise, if $R$ is $\text{strong\_ordering}$, then
   
   \[
   a == b \ ? \ \text{strong\_ordering}::\text{equal} : \\
   a < b \ ? \ \text{strong\_ordering}::\text{less} : \\
   \text{strong\_ordering}::\text{greater}
   \]
5. Otherwise, if $R$ is $\text{weak\_ordering}$, then
   
   \[
   a == b \ ? \ \text{weak\_ordering}::\text{equivalent} : \\
   a < b \ ? \ \text{weak\_ordering}::\text{less} : \\
   \text{weak\_ordering}::\text{greater}
   \]
6. Otherwise (when $R$ is $\text{partial\_ordering}$),
   
   \[
   a == b \ ? \ \text{partial\_ordering}::\text{equivalent} : \\
   a < b \ ? \ \text{partial\_ordering}::\text{less} : \\
   b < a \ ? \ \text{partial\_ordering}::\text{greater} : \\
   \text{partial\_ordering}::\text{unordered}
   \]

[Note 1: A synthesized three-way comparison is ill-formed if overload resolution finds usable candidates that do not otherwise meet the requirements implied by the defined expression. — end note]

Let $R$ be the declared return type of a defaulted three-way comparison operator function, and let $x_i$ be the elements of the expanded list of subobjects for an object $x$ of type $C$.

1. If $R$ is $\text{auto}$, then let $cv_i, R_i$ be the type of the expression $x_i <=> x_i$. The operator function is defined as deleted if that expression is not usable or if $R_i$ is not a comparison category type (17.11.2.1) for any $i$. The return type is deduced as the common comparison type (see below) of $R_0, R_1, \ldots, R_{n-1}$.

2. Otherwise, $R_i$ shall not contain a placeholder type. If the synthesized three-way comparison of type $R$ between any objects $x_i$ and $x_j$ is not defined, the operator function is defined as deleted.

3. The return value $V$ of type $R$ of the defaulted three-way comparison operator function with parameters $x$ and $y$ of the same type is determined by comparing corresponding elements $x_i$ and $y_i$ in the expanded lists of subobjects for $x$ and $y$ (in increasing index order) until the first index $i$ where the synthesized three-way comparison of type $R$ between $x_i$ and $y_i$ yields a result value $v_i$ where $v_i != 0$, contextually converted to $\text{bool}$, yields $\text{true}$; $V$ is a copy of $v_i$. If no such index exists, $V$ is $\text{static\_cast}<R>(\text{std::strong\_ordering}::\text{equal})$.

4. The common comparison type $U$ of a possibly-empty list of $n$ comparison category types $T_0, T_1, \ldots, T_{n-1}$ is defined as follows:
   
   1. If at least one $T_i$ is $\text{std::partial\_ordering}$, $U$ is $\text{std::partial\_ordering}$ (17.11.2.2).
   2. Otherwise, if at least one $T_i$ is $\text{std::weak\_ordering}$, $U$ is $\text{std::weak\_ordering}$ (17.11.2.3).
   3. Otherwise, $U$ is $\text{std::strong\_ordering}$ (17.11.2.4).

   [Note 2: In particular, this is the result when $n$ is 0. — end note]

11.11.4 Secondary comparison operators

A secondary comparison operator is a relational operator (7.6.9) or the $\Rightarrow$ operator. A defaulted operator function (12.6.3) for a secondary comparison operator $@$ shall have a declared return type $\text{bool}$.

The operator function with parameters $x$ and $y$ is defined as deleted if

1. overload resolution (12.4), as applied to $x @$ $y$, does not result in a usable candidate, or
2. the candidate selected by overload resolution is not a rewritten candidate.

Otherwise, the operator function yields $x @$ $y$. The defaulted operator function is not considered as a candidate in the overload resolution for the $@$ operator.

[Example 1:}
struct HasNoLessThan { };  

struct C {
  friend HasNoLessThan operator<=>(const C&, const C&);
  bool operator<(const C&) const = default;  // OK, function is deleted
};

—end example]  

11.12 Free store  
[class.free]  
1 Any allocation function for a class T is a static member (even if not explicitly declared static).
2 [Example 1]:  
  class Arena;
  struct B {
    void* operator new(std::size_t, Arena*);
  };  
  struct D1 : B {
  }

  Arena* ap;
  void foo(int i) {
    new (ap) D1;  // calls B::operator new(std::size_t, Arena*)
    new D1[i];   // calls ::operator new[](std::size_t)
    new D1;      // error: ::operator new[](std::size_t) hidden
  }

  —end example]

3 When an object is deleted with a delete-expression (7.6.2.9), a deallocation function (operator delete() for non-array objects or operator delete[]() for arrays) is (implicitly) called to reclaim the storage occupied by the object (6.7.5.5.3).
4 Class-specific deallocation function lookup is a part of general deallocation function lookup (7.6.2.9) and occurs as follows. If the delete-expression is used to deallocate a class object whose static type has a virtual destructor, the deallocation function is the one selected at the point of definition of the dynamic type’s virtual destructor (11.4.7).\footnote{A similar provision is not needed for the array version of operator delete because 7.6.2.9 requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.} Otherwise, if the delete-expression is used to deallocate an object of class T or array thereof, the deallocation function’s name is looked up in the scope of T. If this lookup fails to find the name, general deallocation function lookup (7.6.2.9) continues. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.
5 Any deallocation function for a class X is a static member (even if not explicitly declared static).

[Example 2]:  
  class X {
    void operator delete(void*);
    void operator delete[](void*, std::size_t);
  };

  class Y {
    void operator delete(void*, std::size_t);
    void operator delete[](void*);
  }

  —end example]

6 Since member allocation and deallocation functions are static they cannot be virtual.

[Note 1: However, when the cast-expression of a delete-expression refers to an object of class type, because the deallocation function actually called is looked up in the scope of the class that is the dynamic type of the object if the destructor is virtual, the effect is the same in that case. For example,

  struct B {
    virtual ~B();
    void operator delete(void*, std::size_t);
  };

\footnote{A similar provision is not needed for the array version of operator delete because 7.6.2.9 requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.}
struct D : B {
    void operator delete(void*);
};

struct E : B {
    void log_deletion();
    void operator delete(E *p, std::destroying_delete_t) {
        p->log_deletion();
        p->~E();
        ::operator delete(p);
    }
};

void f() {
    B* bp = new D;
    delete bp;    // uses D::operator delete(void*)
    bp = new E;
    delete bp;    // uses E::operator delete(E*, std::destroying_delete_t)
}

Here, storage for the object of class D is deallocated by D::operator delete(), and the object of class E is destroyed and its storage is deallocated by E::operator delete(), due to the virtual destructor. —end note]

[Note 2: Virtual destructors have no effect on the deallocation function actually called when the cast-expression of a delete-expression refers to an array of objects of class type. For example,

    struct B {
        virtual ~B();
        void operator delete[](void*, std::size_t);
    };

    struct D : B {
        void operator delete[](void*, std::size_t);
    };

    void f(int i) {
        D* dp = new D[i];
        delete [] dp;    // uses D::operator delete[](void*, std::size_t)
        B* bp = new D[i];
        delete[] bp;    // undefined behavior
    }

—end note]

7 Access to the deallocation function is checked statically.

[Note 3: Hence, even though a different one might actually be executed, the statically visible deallocation function is required to be accessible. —end note]

[Example 3: For the call on line “// 1” above, if B::operator delete() had been private, the delete expression would have been ill-formed. —end example]

8 [Note 4: If a deallocation function has no explicit noexcept-specifier, it has a non-throwing exception specification (14.5). —end note]
12 Overloading

12.1 Preamble

When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded, and the declarations are called overloaded declarations. Only function and function template declarations can be overloaded; variable and type declarations cannot be overloaded.

When a function name is used in a call, which function declaration is being referenced and the validity of the call are determined by comparing the types of the arguments at the point of use with the types of the parameters in the declarations that are visible at the point of use. This function selection process is called overload resolution and is defined in 12.4.

Example 1:

```c
double abs(double);
int abs(int);
abs(1);        // calls abs(int);
abs(1.0);      // calls abs(double);
```

12.2 Overloadable declarations

Not all function declarations can be overloaded. Those that cannot be overloaded are specified here. A program is ill-formed if it contains two such non-overloadable declarations in the same scope.

Note 1: This restriction applies to explicit declarations in a scope, and between such declarations and declarations made through a using-declaration (9.9). It does not apply to sets of functions fabricated as a result of name lookup (e.g., because of using-directives) or overload resolution (e.g., for operator functions).

Certain function declarations cannot be overloaded:

(2.1) Function declarations that differ only in the return type, the exception specification (14.5), or both cannot be overloaded.

(2.2) Member function declarations with the same name, the same parameter-type-list (9.3.4.6), and the same trailing requires-clause (if any) cannot be overloaded if any of them is a static member function declaration (11.4.9). Likewise, member function template declarations with the same name, the same parameter-type-list, the same trailing requires-clause (if any), and the same template-head cannot be overloaded if any of them is a static member function template declaration. The types of the implicit object parameters constructed for the member functions for the purpose of overload resolution (12.4.2) are not considered when comparing parameter-type-lists for enforcement of this rule. In contrast, if there is no static member function declaration among a set of member function declarations with the same name, the same parameter-type-list, and the same trailing requires-clause (if any), then these member function declarations can be overloaded if they differ in the type of their implicit object parameter.

Example 1: The following illustrates this distinction:

```c
class X {
    static void f();
    void f();          // error
    void f() const;    // error
    void f() const volatile; // error
    void g();
    void g() const;    // OK: no static g
    void g() const volatile; // OK: no static g
};
```

Member function declarations with the same name, the same parameter-type-list (9.3.4.6), and the same trailing requires-clause (if any), as well as member function template declarations with the same name,
the same parameter-type-list, the same trailing requires-clause (if any), and the same template-head, cannot be overloaded if any of them, but not all, have a ref-qualifier (9.3.4.6).

[Example 2:

```cpp
class Y {
    void h() &;
    void h() const &;   // OK
    void h() &&;        // OK, all declarations have a ref-qualifier
    void i() &;
    void i() const;     // error: prior declaration of i has a ref-qualifier
};
```
—end example]

3 [Note 2: As specified in 9.3.4.6, function declarations that have equivalent parameter declarations and requires-clauses, if any (13.5.3), declare the same function and therefore cannot be overloaded:

(3.1) — Parameter declarations that differ only in the use of equivalent typedef “types” are equivalent. A typedef is not a separate type, but only a synonym for another type (9.2.4).

[Example 3:

```cpp
typedef int Int;
void f(int i);
void f(Int i);  // OK: redeclaration of f(int)
void f(int i) { /* ... */ }
void f(Int i) { /* ... */ }  // error: redefinition of f(int)
```
—end example]

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded function declarations.

[Example 4:

```cpp
enum E { a };
void f(int i) { /* ... */ }
void f(E i) { /* ... */ }
```
—end example]

(3.2) — Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration (9.3.4.6). Only the second and subsequent array dimensions are significant in parameter types (9.3.4.5).

[Example 5:

```cpp
int f(char*);
int f(char[]);  // same as f(char*)
int f(char[7]);  // same as f(char*)
int f(char[9]);  // same as f(char*)

int g(char*)[10]);
int g(char[5]) [10]);  // same as g(char*)(10));
int g(char[7]) [10]);  // same as g(char*)[10]);
int g(char*)[20]);  // different from g(char*)[10]);
```
—end example]

(3.3) — Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent. That is, the function type is adjusted to become a pointer to function type (9.3.4.6).

[Example 6:

```cpp
void h(int());
void h(int (*))();  // redeclaration of h(int())
void h(int x()); { }  // definition of h(int())
void h(int (*x)()) { }  // error: redefinition of h(int())
```
—end example]
Parameter declarations that differ only in the presence or absence of `const` and/or `volatile` are equivalent. That is, the `const` and `volatile` type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called.

**Example 7:**
```c
typedef const int cInt;

int f (int);
int f (const int);    // redeclaration of f(int)
int f (int) { /* ... */ }    // definition of f(int)
int f (cInt) { /* ... */ }    // error: redefinition of f(int)
```

Only the `const` and `volatile` type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; `const` and `volatile` type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type `T`, “pointer to `T`”, “pointer to `const T`”, and “pointer to `volatile T`” are considered distinct parameter types, as are “reference to `T`”, “reference to `const T`”, and “reference to `volatile T`”.

Two parameter declarations that differ only in their default arguments are equivalent.

**Example 8:** Consider the following:
```c
void f (int i, int j);
void f (int i, int j = 99);    // OK: redeclaration of f(int, int)
void f (int i = 88, int j);    // OK: redeclaration of f(int, int)
void f ();                     // OK: overloaded declaration of f

void prog () {
    f (1, 2);    // OK: call f(int, int)
    f (1);       // OK: call f(int, int)
    f ();        // error: f(int, int) or f()?
}
```

12.3 Declaration matching

Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations and equivalent (13.7.7.2) trailing requires-clauses, if any (9.3).

**Note 1:** Since a `constraint-expression` is an unevaluated operand, equivalence compares the expressions without evaluating them.

**Example 1:**
```c
template<int I> concept C = true;

template<typename T> struct A {
    void f() requires C<42>;    // #1
    void f() requires true;     // OK, different functions
};
```

A function member of a derived class is not in the same scope as a function member of the same name in a base class.

**Example 2:**
```c
struct B {
    int f(int);
};
```

120) When a parameter type includes a function type, such as in the case of a parameter type that is a pointer to function, the `const` and `volatile` type-specifiers at the outermost level of the parameter type specifications for the inner function type are also ignored.
struct D : B {
    int f(const char*);
};

Here D::f(const char*) hides B::f(int) rather than overloading it.

void h(D* pd) {
    pd->f(1); // error: D::f(const char*) hides B::f(int)
    pd->B::f(1); // OK
    pd->f("Ben"); // OK, calls D::f
}

—end example|

2 A locally declared function is not in the same scope as a function in a containing scope.

Example 3:

void f(const char*);

void g() {
    extern void f(int);
    f("asdf"); // error: f(int) hides f(const char*)
    // so there is no f(const char*) in this scope
}

void caller () {
    extern void callee(int, int);
    {
        extern void callee(int); // hides callee(int, int)
        callee(88, 99); // error: only callee(int) in scope
    }
}

—end example|

3 Different versions of an overloaded member function can be given different access rules.

Example 4:

class buffer {
private:
    char* p;
    int size;
protected:
    buffer(int s, char* store) { size = s; p = store; }
public:
    buffer(int s) { p = new char[size = s]; }
};

—end example|

12.4 Overload resolution

12.4.1 General

Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of candidate functions that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the parameter-type-list of the candidate function, how well (for non-static member functions) the object matches the implicit object parameter, and certain other properties of the candidate function.

[Note 1: The function selected by overload resolution is not guaranteed to be appropriate for the context. Other restrictions, such as the accessibility of the function, can make its use in the calling context ill-formed. — end note]

2 Overload resolution selects the function to call in seven distinct contexts within the language:

(2.1) invocation of a function named in the function call syntax (12.4.2.2.2);
(2.2) invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax (12.4.2.2.3);
(2.3) invocation of the operator referenced in an expression (12.4.2.3);
Each of these contexts defines the set of candidate functions and the list of arguments in its own unique way. But, once the candidate functions and argument lists have been identified, the selection of the best function is the same in all cases:

(2.8) First, a subset of the candidate functions (those that have the proper number of arguments and meet certain other conditions) is selected to form a set of viable functions (12.4.3).

(2.9) Then the best viable function is selected based on the implicit conversion sequences (12.4.4.2) needed to match each argument to the corresponding parameter of each viable function.

If a best viable function exists and is unique, overload resolution succeeds and produces it as the result. Otherwise overload resolution fails and the invocation is ill-formed. When overload resolution succeeds, and the best viable function is not accessible (11.9) in the context in which it is used, the program is ill-formed.

Overload resolution results in a usable candidate if overload resolution succeeds and the selected candidate is either not a function (12.7), or is a function that is not deleted and is accessible from the context in which overload resolution was performed.
Because other than in list-initialization only one user-defined conversion is allowed in an implicit conversion sequence, special rules apply when selecting the best user-defined conversion (12.4.4, 12.4.4.2).

Example 2:
```cpp
class T {
    public:
        T();
};

class C : T {
    public:
        C(int);
    };

T a = 1;  // error: no viable conversion (T(C(1)) not considered)
```

In each case where a candidate is a function template, candidate function template specializations are generated using template argument deduction (13.10.4, 13.10.3). If a constructor template or conversion function template has an explicit-specifier whose constant-expression is value-dependent (13.8.3), template argument deduction is performed first and then, if the context requires a candidate that is not explicit and the generated specialization is explicit (9.2.3), it will be removed from the candidate set. Those candidates are then handled as candidate functions in the usual way. A given name can refer to one or more function templates and also to a set of non-template functions. In such a case, the candidate functions generated from each function template are combined with the set of non-template candidate functions.

A defaulted move special member function (11.4.5.3, 11.4.6) that is defined as deleted is excluded from the set of candidate functions in all contexts. A constructor inherited from class type C (11.10.4) that has a first parameter of type “reference to cv1 P” (including such a constructor instantiated from a template) is excluded from the set of candidate functions when constructing an object of type cv2 D if the argument list has exactly one argument and C is reference-related to P and P is reference-related to D.

Example 3:
```cpp
struct A {
    A();                  // #1
    A(A &&);             // #2
    template<typename T> A(T &&);  // #3
};

struct B : A {
    using A::A;
    B(const B &);        // #4
    B(B &&) = default;   // #5, implicitly deleted
}

struct X { X(X &&) = delete; } x;

struct Y {
    Y(Y &&) = default;
}

extern B b1;
B b2 = static_cast<B&&>(b1); // calls #4: #1 is not viable, #2, #3, and #5 are not candidates
struct C {
    operator B&&();
};
B b3 = C();                  // calls #4
```

12.4.2.2 Function call syntax

12.4.2.2.1 General

In a function call (7.6.1.3)

postfix-expression ( expression-listopt )

121) The process of argument deduction fully determines the parameter types of the function template specializations, i.e., the parameters of function template specializations contain no template parameter types. Therefore, except where specified otherwise, function template specializations and non-template functions (9.3.4.6) are treated equivalently for the remainder of overload resolution.
In addition, for each non-explicit conversion function declared in 12.4.2.2.2, If the postfix-expression denotes an object of class type, overload resolution is applied as specified in 12.4.2.2.3.

If the postfix-expression is the address of an overload set, overload resolution is applied using that set as described above. If the function selected by overload resolution is a non-static member function, the program is ill-formed.

[Note 1: The resolution of the address of an overload set in other contexts is described in 12.5. — end note]

12.4.2.2.2 Call to named function [over.call.func]

Of interest in 12.4.2.2.2 are only those function calls in which the postfix-expression ultimately contains a name that denotes one or more functions. Such a postfix-expression, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

postfix-expression:
  postfix-expression . id-expression
  postfix-expression -> id-expression
  primary-expression

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

In qualified function calls, the name to be resolved is an id-expression and is preceded by an -> or . operator. Since the construct A->B is generally equivalent to (*A).B, the rest of Clause 12 assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the . operator. Furthermore, Clause 12 assumes that the postfix-expression that is the left operand of the . operator has type “cv T” where T denotes a class. Under this assumption, the id-expression in the call is looked up as a member function of T following the rules for looking up names in classes (11.8). The function declarations found by that lookup constitute the set of candidate functions. The argument list is the expression-list in the call augmented by the addition of the left operand of the . operator in the normalized member function call as the implied object argument (12.4.2).

In unqualified function calls, the name is not qualified by an -> or . operator and has the more general form of a primary-expression. The name is looked up in the context of the function call following the normal rules for name lookup in expressions (6.5). The function declarations found by that lookup constitute the set of candidate functions. Because of the rules for name lookup, the set of candidate functions consists (1) entirely of non-member functions or (2) entirely of member functions of some class T. In case (1), the argument list is the same as the expression-list in the call. In case (2), the argument list is the expression-list in the call augmented by the addition of an implied object argument as in a qualified function call. If the keyword this (11.4.3.2) is in scope and refers to class T, or a derived class of T, then the implied object argument is (*this). If the keyword this is not in scope or refers to another class, then a contrived object argument of type T becomes the implied object argument. If the argument list is augmented by a contrived object and overload resolution selects one of the non-static member functions of T, the call is ill-formed.

12.4.2.2.3 Call to object of class type [over.call.object]

If the postfix-expression E in the function call syntax evaluates to a class object of type “cv T”, then the set of candidate functions includes at least the function call operators of T. The function call operators of T are obtained by ordinary lookup of the name operator() in the context of (E).operator().

In addition, for each non-explicit conversion function declared in T of the form

operator conversion-type-id ( ) cv-qualifier-seqopt ref-qualifieropt noexcept-specifieropt attribute-specifier-seqopt ;

where the optional cv-qualifier-seq is the same cv-qualification as, or a greater cv-qualification than, cv, and where conversion-type-id denotes the type “pointer to function of (P1,...,Pn) returning R”, or the type “reference to pointer to function of (P1,...,Pn) returning R”, or the type “reference to function of (P1,...,Pn) returning R”, a surrogate call function with the unique name call-function and having the form

R call-function ( conversion-type-id F, P1 a1, ..., Pn an ) { return F ( a1, ..., an ); }

[122] Note that cv-qualifiers on the type of objects are significant in overload resolution for both glvalue and class prvalue objects.

[123] An implied object argument is contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function.
is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each non-explicit conversion function declared in a base class of $T$ provided the function is not hidden within $T$ by another intervening declaration.\textsuperscript{124}

3 The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument (E).

[Note 1: When comparing the call against the function call operators, the implied object argument is compared against the implicit object parameter of the function call operator. When comparing the call against a surrogate call function, the implied object argument is compared against the first parameter of the surrogate call function. The conversion function from which the surrogate call function was derived will be used in the conversion sequence for that parameter since it converts the implied object argument to the appropriate function pointer or reference required by that first parameter. —end note]

[Example 1:

```c
int f1(int);
int f2(float);
typed int (*fp1)(int);
typed int (*fp2)(float);
struct A {
  operator fp1() { return f1; }
  operator fp2() { return f2; }
} a;
int i = a(1); // calls f1 via pointer returned from conversion function
```

—end example]

12.4.2.3 Operators in expressions

If no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to 7.6.

[Note 1: Because .., *, and :: cannot be overloaded, these operators are always built-in operators interpreted according to 7.6. ?: cannot be overloaded, but the rules in this subclause are used to determine the conversions to be applied to the second and third operands when they have class or enumeration type (7.6.16). —end note]

[Example 1:

```c
struct String {
  String (const String&);
  String (const char*);
  operator const char* () {
  }
};
String operator + (const String&, const String&);

void f() {
  const char* p= "one" + "two"; // error: cannot add two pointers; overloaded operator+ not considered
  // because neither operand has class or enumeration type
  int I = 1 + 1; // always evaluates to 2 even if class or enumeration types exist
  // that would perform the operation.
}
```

—end example]

If either operand has a type that is a class or an enumeration, a user-defined operator function can be declared that implements this operator or a user-defined conversion can be necessary to convert the operand to a type that is appropriate for a built-in operator. In this case, overload resolution is used to determine which operator function or built-in operator is to be invoked to implement the operator. Therefore, the operator notation is first transformed to the equivalent function-call notation as summarized in Table 15 (where $@$ denotes one of the operators covered in the specified subclause). However, the operands are sequenced in the order prescribed for the built-in operator (7.6).

3 For a unary operator $@$ with an operand of type $cv1$ $T1$, and for a binary operator $@$ with a left operand of type $cv1$ $T1$ and a right operand of type $cv2$ $T2$, four sets of candidate functions, designated member candidates, non-member candidates, built-in candidates, and rewritten candidates, are constructed as follows:

\textsuperscript{124} Note that this construction can yield candidate call functions that cannot be differentiated one from the other by overload resolution because they have identical declarations or differ only in their return type. The call will be ambiguous if overload resolution cannot select a match to the call that is uniquely better than such undifferentiable functions.
Table 15: Relationship between operator and function call notation

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Expression</th>
<th>As member function</th>
<th>As non-member function</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.6.2</td>
<td>@a</td>
<td>(a).operator@ ()</td>
<td>operator@(a)</td>
</tr>
<tr>
<td>12.6.3</td>
<td>a@b</td>
<td>(a).operator@ (b)</td>
<td>operator@(a, b)</td>
</tr>
<tr>
<td>12.6.3.2</td>
<td>a=b</td>
<td>(a).operator= (b)</td>
<td></td>
</tr>
<tr>
<td>12.6.5</td>
<td>a[b]</td>
<td>(a).operator[] (b)</td>
<td></td>
</tr>
<tr>
<td>12.6.6</td>
<td>a-&gt;</td>
<td>(a).operator-&gt;()</td>
<td></td>
</tr>
<tr>
<td>12.6.7</td>
<td>a@0</td>
<td>(a).operator@ (0)</td>
<td>operator@(a, 0)</td>
</tr>
</tbody>
</table>

(3.1) If T1 is a complete class type or a class currently being defined, the set of member candidates is the result of the qualified lookup of T1::operator@ (12.4.2.2.2); otherwise, the set of member candidates is empty.

(3.2) The set of non-member candidates is the result of the unqualified lookup of operator@ in the context of the expression according to the usual rules for name lookup in unqualified function calls (6.5.3) except that all member functions are ignored. However, if no operand has a class type, only those non-member functions in the lookup set that have a first parameter of type T1 or “reference to cv T1”, when T1 is an enumeration type, or (if there is a right operand) a second parameter of type T2 or “reference to cv T2”, when T2 is an enumeration type, are candidate functions.

(3.3) For the operator , , the unary operator & , or the operator ->, the built-in candidates set is empty. For all other operators, the built-in candidates include all of the candidate operator functions defined in 12.7 that, compared to the given operator,

(3.3.1) have the same operator name, and

(3.3.2) accept the same number of operands, and

(3.3.3) accept operand types to which the given operand or operands can be converted according to 12.4.4.2, and

(3.3.4) do not have the same parameter-type-list as any non-member candidate that is not a function template specialization.

(3.4) The rewritten candidate set is determined as follows:

(3.4.1) For the relational (7.6.9) operators, the rewritten candidates include all non-rewritten candidates for the expression x <=> y.

(3.4.2) For the relational (7.6.9) and three-way comparison (7.6.8) operators, the rewritten candidates also include a synthesized candidate, with the order of the two parameters reversed, for each non-rewritten candidate for the expression y <=> x.

(3.4.3) For the != operator (7.6.10), the rewritten candidates include all non-rewritten candidates for the expression x == y.

(3.4.4) For the equality operators, the rewritten candidates also include a synthesized candidate, with the order of the two parameters reversed, for each non-rewritten candidate for the expression y == x.

(3.4.5) For all other operators, the rewritten candidate set is empty.

[Note 2: A candidate synthesized from a member candidate has its implicit object parameter as the second parameter, thus implicit conversions are considered for the first, but not for the second, parameter. —end note]

4 For the built-in assignment operators, conversions of the left operand are restricted as follows:

(4.1) no temporaries are introduced to hold the left operand, and

(4.2) no user-defined conversions are applied to the left operand to achieve a type match with the left-most parameter of a built-in candidate.

5 For all other operators, no such restrictions apply.

6 The set of candidate functions for overload resolution for some operator @ is the union of the member candidates, the non-member candidates, the built-in candidates, and the rewritten candidates for that operator @.
The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 12.4.3 and 12.4.4.125

Example 2:

```c
struct A {
    operator int();
};
A operator+(const A&, const A&);
void m() {
    A a, b;
    a + b;                  // operator+(a, b) chosen over int(a) + int(b)
}  
@end example
```

If a rewritten operator\(<<\) candidate is selected by overload resolution for an operator \(@\), \(x \@ y\) is interpreted as \(0 \@ (y \<\> x)\) if the selected candidate is a synthesized candidate with reversed order of parameters, or \((x \<\> y) \@ 0\) otherwise, using the selected rewritten operator\(<<\) candidate. Rewritten candidates for the operator \(@\) are not considered in the context of the resulting expression.

If a rewritten operator\(==\) candidate is selected by overload resolution for an operator \@\, its return type shall be \cv bool, and \(x \@ y\) is interpreted as:

\[(9.1)\] if \@ is \(!=\) and the selected candidate is a synthesized candidate with reversed order of parameters, \(! (y == x)\),

\[(9.2)\] otherwise, if \@ is \(!=\), \(! (x == y)\),

\[(9.3)\] otherwise (when \@ is \(==\)), \(y == x\),

in each case using the selected rewritten operator\(==\) candidate.

If a built-in candidate is selected by overload resolution, the operands of class type are converted to the types of the corresponding parameters of the selected operation function, except that the second standard conversion sequence of a user-defined conversion sequence (12.4.4.2.3) is not applied. Then the operator is treated as the corresponding built-in operator and interpreted according to 7.6.

Example 3:

```c
struct X {
    operator double();
};
struct Y {
    operator int*();
};

int *a = Y() + 100.0;   // error: pointer arithmetic requires integral operand
int *b = Y() + X();    // error: pointer arithmetic requires integral operand
@end example
```

The second operand of operator \(~\) is ignored in selecting an operator\(~\) function, and is not an argument when the operator\(~\) function is called. When operator\(~\) returns, the operator \(~\) is applied to the value returned, with the original second operand.\[126\]

If the operator is the operator \(\&\), the unary operator \&, or the operator \(~\), and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to 7.6.

Note 3: The lookup rules for operators in expressions are different than the lookup rules for operator function names in a function call, as shown in the following example:

```c
struct A {
};
void operator + (A, A);

struct B {
    void operator + (B);
```

---

125) If the set of candidate functions is empty, overload resolution is unsuccessful.
126) If the value returned by the operator\(~\) function has class type, this can result in selecting and calling another operator\(~\) function. The process repeats until an operator\(~\) function returns a value of non-class type.
```cpp
void f () {
};
A a;

void B::f() {
    operator+ (a,a); // error: global operator hidden by member
    a + a; // OK: calls global operator+
}
```

12.4.2.4 Initialization by constructor

When objects of class type are direct-initialized (9.4), copy-initialized from an expression of the same or a derived class type (9.4), or default-initialized (9.4), overload resolution selects the constructor. For direct-initialization or default-initialization that is not in the context of copy-initialization, the candidate functions are all the constructors of the class of the object being initialized. For copy-initialization (including default initialization in the context of copy-initialization), the candidate functions are all the converting constructors (11.4.8.2) of that class. The argument list is the expression-list or assignment-expression of the initializer.

12.4.2.5 Copy-initialization of class by user-defined conversion

Under the conditions specified in 9.4, as part of a copy-initialization of an object of class type, a user-defined conversion can be invoked to convert an initializer expression to the type of the object being initialized. Overload resolution is used to select the user-defined conversion to be invoked.

[Note 1: The conversion performed for indirect binding to a reference to a possibly cv-qualified class type is determined in terms of a corresponding non-reference copy-initialization. — end note]

Assuming that "cv1 T" is the type of the object being initialized, with T a class type, the candidate functions are selected as follows:

1. The converting constructors (11.4.8.2) of T are candidate functions.
2. When the type of the initializer expression is a class type “cv S”, the non-explicit conversion functions of S and its base classes are considered. When initializing a temporary object (11.4) to be bound to the first parameter of a constructor where the parameter is of type “reference to cv2 T” and the constructor is called with a single argument in the context of direct-initialization of an object of type “cv3 T”, explicit conversion functions are also considered. Those that are not hidden within S and yield a type whose cv-unqualified version is the same type as T or is a derived class thereof are candidate functions. A call to a conversion function returning “reference to X” is a glvalue of type X, and such a conversion function is therefore considered to yield X for this process of selecting candidate functions.

In both cases, the argument list has one argument, which is the initializer expression.

[Note 2: This argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions. — end note]

12.4.2.6 Initialization by conversion function

Under the conditions specified in 9.4, as part of an initialization of an object of non-class type, a conversion function can be invoked to convert an initializer expression of class type to the type of the object being initialized. Overload resolution is used to select the conversion function to be invoked. Assuming that “cv1 T” is the type of the object being initialized, and “cv S” is the type of the initializer expression, with S a class type, the candidate functions are selected as follows:

1. The conversion functions of S and its base classes are considered. Those non-explicit conversion functions that are not hidden within S and yield type T or a type that can be converted to type T via a standard conversion sequence (12.4.4.2.2) are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within S and yield type T or a type that can be converted to type T with a qualification conversion (7.3.6) are also candidate functions. Conversion functions that return a cv-qualified type are considered to yield the cv-unqualified version of that type for this process of selecting candidate functions. A call to a conversion function returning “reference to X” is a glvalue of type X, and such a conversion function is therefore considered to yield X for this process of selecting candidate functions.
2 The argument list has one argument, which is the initializer expression.

[Note 1: This argument will be compared against the implicit object parameter of the conversion functions. — end note]

12.4.2.7 Initialization by conversion function for direct reference binding  [over.match.ref]

Under the conditions specified in 9.4.4, a reference can be bound directly to the result of applying a conversion function to an initializer expression. Overload resolution is used to select the conversion function to be invoked. Assuming that “reference to \( T \)” is the type of the reference being initialized, and “\( CV S \)” is the type of the initializer expression, with \( S \) a class type, the candidate functions are selected as follows:

1) The conversion functions of \( S \) and its base classes are considered. Those non-explicit conversion functions that are not hidden within \( S \) and yield type “value reference to \( CV2 T2 \)” (when initializing an Ivalue reference or an rvalue reference to function) or “\( CV2 T2 \)” or “rvalue reference to \( CV2 T2 \)” (when initializing an rvalue reference or an Ivalue reference to function), where “\( CV1 T \)” is reference-compatible (9.4.4) with “\( CV2 T2 \)”, are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within \( S \) and yield type “value reference to \( CV2 T2 \)” (when initializing an Ivalue reference or an rvalue reference to function) or “rvalue reference to \( CV2 T2 \)” (when initializing an rvalue reference or an Ivalue reference to function), where \( T2 \) is the same type as \( T \) or can be converted to type \( T \) with a qualification conversion (7.3.6), are also candidate functions.

2) The argument list has one argument, which is the initializer expression.

[Note 1: This argument will be compared against the implicit object parameter of the conversion functions. — end note]

12.4.2.8 Initialization by list-initialization  [over.match.list]

When objects of non-aggregate class type \( T \) are list-initialized such that 9.4.5 specifies that overload resolution is performed according to the rules in this subclause or when forming a list-initialization sequence according to 12.4.4.2.6, overload resolution selects the constructor in two phases:

1) If the initializer list is not empty or \( T \) has no default constructor, overload resolution is first performed where the candidate functions are the initializer-list constructors (9.4.5) of the class \( T \) and the argument list consists of the initializer list as a single argument.

2) Otherwise, or if no viable initializer-list constructor is found, overload resolution is performed again, where the candidate functions are all the constructors of the class \( T \) and the argument list consists of the elements of the initializer list.

In copy-list-initialization, if an explicit constructor is chosen, the initialization is ill-formed.

[Note 1: This differs from other situations (12.4.2.4, 12.4.2.5), where only converting constructors are considered for copy-initialization. This restriction only applies if this initialization is part of the final result of overload resolution. — end note]

12.4.2.9 Class template argument deduction  [over.match.class.deduct]

When resolving a placeholder for a deduced class type (9.2.9.7) where the template-name names a primary class template \( C \), a set of functions and function templates, called the guides of \( C \), is formed comprising:

1) If \( C \) is defined, for each constructor of \( C \), a function template with the following properties:

1.1) The template parameters are the template parameters of \( C \) followed by the template parameters (including default template arguments) of the constructor, if any.

1.2) The types of the function parameters are those of the constructor.

1.3) The return type is the class template specialization designated by \( C \) and template arguments corresponding to the template parameters of \( C \).

2) If \( C \) is not defined or does not declare any constructors, an additional function template derived as above from a hypothetical constructor \( C() \).

3) An additional function template derived as above from a hypothetical constructor \( C(C) \), called the copy deduction candidate.

4) For each deduction-guide, a function or function template with the following properties:

4.1) The template parameters, if any, and function parameters are those of the deduction-guide.

4.2) The return type is the simple-template-id of the deduction-guide.
In addition, if C is defined and its definition satisfies the conditions for an aggregate class (9.4.2) with the assumption that any dependent base class has no virtual functions and no virtual base classes, and the initializer is a non-empty braced-init-list or parenthesized expression-list, and there are no deduction-guides for C, the set contains an additional function template, called the aggregate deduction candidate, defined as follows. Let \(x_1, \ldots, x_n\) be the elements of the initializer-list or designated-initializer-list of the braced-init-list, or of the expression-list. For each \(x_i\), let \(e_i\) be the corresponding aggregate element of C or of one of its (possibly recursive) subaggregates that would be initialized by \(x_i\) (9.4.2) if

1.5  --- brace elision is not considered for any aggregate element that has a dependent non-array type or an array type with a value-dependent bound, and

1.6  --- each non-trailing aggregate element that is a pack expansion is assumed to correspond to no elements of the initializer list, and

1.7  --- a trailing aggregate element that is a pack expansion is assumed to correspond to all remaining elements of the initializer list (if any).

If there is no such aggregate element \(e_i\) for any \(x_i\), the aggregate deduction candidate is not added to the set. The aggregate deduction candidate is derived as above from a hypothetical constructor \(C(T_1, \ldots, T_n)\), where

1.8  --- if \(e_i\) is of array type and \(x_i\) is a braced-init-list or string-literal, \(T_i\) is an rvalue reference to the declared type of \(e_i\), and

1.9  --- otherwise, \(T_i\) is the declared type of \(e_i\),

except that additional parameter packs of the form \(P_j, \ldots\) are inserted into the parameter list in their original aggregate element position corresponding to each non-trailing aggregate element of type \(P_j\) that was skipped because it was a parameter pack, and the trailing sequence of parameters corresponding to a trailing aggregate element that is a pack expansion (if any) is replaced by a single parameter of the form \(T_n, \ldots\).

2 When resolving a placeholder for a deduced class type (9.2.9.3) where the template-name names an alias template \(A\), the defining-type-id of \(A\) must be of the form

\[
\text{typename}_{\text{opt}} \ \text{nested-name-specifier}_{\text{opt}} \ \text{template}_{\text{opt}} \ \text{simple-template-id}
\]

as specified in 9.2.9.3. The guides of \(A\) are the set of functions or function templates formed as follows. For each function or function template \(f\) in the guides of the template named by the simple-template-id of the defining-type-id, the template arguments of the return type of \(f\) are deduced from the defining-type-id of \(A\) according to the process in 13.10.3.6 with the exception that deduction does not fail if not all template arguments are deduced. Let \(g\) denote the result of substituting these deductions into \(f\). If substitution succeeds, form a function or function template \(f'\) with the following properties and add it to the set of guides of \(A\):

2.1  --- The function type of \(f'\) is the function type of \(g\).

2.2  --- If \(f\) is a function template, \(f'\) is a function template whose template parameter list consists of all the template parameters of \(A\) (including their default template arguments) that appear in the above deductions or (recursively) in their default template arguments, followed by the template parameters of \(f\) that were not deduced (including their default template arguments), otherwise \(f'\) is not a function template.

2.3  --- The associated constraints (13.5.3) are the conjunction of the associated constraints of \(g\) and a constraint that is satisfied if and only if the arguments of \(A\) are deducible (see below) from the return type.

2.4  --- If \(f\) is a copy deduction candidate, then \(f'\) is considered to be so as well.

2.5  --- If \(f\) was generated from a deduction-guide (13.7.2.3), then \(f'\) is considered to be so as well.

2.6  --- The explicit-specifier of \(f'\) is the explicit-specifier of \(g\) (if any).

3 The arguments of a template \(A\) are said to be deducible from a type \(T\) if, given a class template

\[
\text{template} <\text{typename}> \ \text{class} \ AA;
\]

with a single partial specialization whose template parameter list is that of \(A\) and whose template argument list is a specialization of \(A\) with the template argument list of \(A\) (13.8.3.2), \(AA<T>\) matches the partial specialization.

4 Initialization and overload resolution are performed as described in 9.4 and 12.4.2.4, 12.4.2.5, or 12.4.2.8 (as appropriate for the type of initialization performed) for an object of a hypothetical class type, where the guides of the template named by the placeholder are considered to be the constructors of that class type for
the purpose of forming an overload set, and the initializer is provided by the context in which class template argument deduction was performed. The following exceptions apply:

(4.1) — The first phase in 12.4.2.8 (considering initializer-list constructors) is omitted if the initializer list consists of a single expression of type cv U, where U is, or is derived from, a specialization of the class template directly or indirectly named by the placeholder.

(4.2) — During template argument deduction for the aggregate deduction candidate, the number of elements in a trailing parameter pack is only deduced from the number of remaining function arguments if it is not otherwise deduced.

If the function or function template was generated from a constructor or deduction-guide that had an explicit-specifier, each such notional constructor is considered to have that same explicit-specifier. All such notional constructors are considered to be public members of the hypothetical class type.

Example 1:

```cpp
template <class T> struct A {
    explicit A(const T&, ...) noexcept;       // #1
    A(T&&, ...);                           // #2
};

int i;
A a1 = { i, i };                      // error: explicit constructor #1 selected in copy-list-initialization during deduction,
                                        // cannot deduce from non-forwarding rvalue reference in #2

A a2(i, i);                           // OK, #1 deduces to A<int> and also initializes
A a3(0, i);                           // OK, #2 deduces to A<int> and also initializes
A a4 = {0, i};                        // OK, #2 deduces to A<int> and also initializes

template <class T> A(const T&, const T&) -> A<T&>;       // #3
template <class T> explicit A(T&&, T&&) -> A<T>;         // #4

A a5 = {0, 1};                       // error: explicit deduction guide #4 selected in copy-list-initialization during deduction
A a6(0,1);                          // OK, #4 deduces to A<int> and #2 initializes
A a7 = {0, i};                       // error: #3 deduces to A<int&>, #1 and #2 declare same constructor
A a8(0,i);                          // error: #3 deduces to A<int&>, #1 and #2 declare same constructor

template <class T> struct B {
    template <class U> using TA = T;
    template <class U> B(U, TA<U>);
};

B b{(int*)0, (char*)0};               // OK, deduces B<char>

```
D d1 = {1, 2}; // error: deduction failed
D d2 = {1, 2, 3}; // OK, braces elided, deduces D<int>

template <typename T>
struct E {
    T t;
    decltype(t) t2;
};
E e1 = {1, 2}; // OK, deduces E<int>

template <typename... T>
struct Types {}

struct X {};
struct Y {};
struct Z {};
struct W { operator Y(); }
F f1 = {Types<X, Y, Z>{}, {}, {}}; // OK, F<X, Y, Z> deduced
F f2 = {Types<X, Y, Z>{}, X{}, Y{}}; // OK, F<X, Y, Z> deduced
F f3 = {Types<X, Y, Z>{}, X{}, W{}}; // error: conflicting types deduced; operator Y not considered
—end example

Example 2:

template <class T, class U> struct C {
    C(T, U);
};

template<class T, class U>
C(T, U) -> C<T, std::type_identity_t<U>>; // #2

template<class V> using A = C<V *, V *>

int i{};
double d{};

A a1(&i, &i); // deduces A<int>
A a2(i, i); // error: cannot deduce V * from i
A a3(&i, &d); // error: #1: cannot deduce (V*, V*) from (int *, double *)
    // #2: cannot deduce A<V> from C<int *, double *>
B b1(&i, &i); // deduces B<int>
B b2(&d, &d); // error: cannot deduce B<W> from C<double *, double *>

Possible exposition-only implementation of the above procedure:

// The following concept ensures a specialization of A is deduced.
template <class> class AA;
template <class V> class AA<A<V>> { };
template <class T> concept deduces_A = requires { sizeof(AA<T>); };

// f1 is formed from the constructor #1 of C, generating the following function template
template<T, U>
auto f1(T, U) -> C<T, U>;

// Deducing arguments for C<T, U> from C<V *, V *> deduces T as V * and U as V *;
// f1' is obtained by transforming f1 as described by the above procedure.
template<class V> requires deduces_A<C<V *, V *>>
    auto f1_prime(V *, V *) -> C<V *, V *>;

// f2 is formed from the deduction-guide #2 of C
template<class T, class U> auto f2(T, U) -> C<T, std::type_identity_t<U>>;
// Deducing arguments for C<T, std::type_identity_t<U>> from C<V *, V*> deduces T as V *;
// f2' is obtained by transforming f2 as described by the above procedure.
template<class V, class U>
  requires deduces_A<C<V *, std::type_identity_t<U>>>
  auto f2_prime(V *, U) -> C<V *, std::type_identity_t<U>>;

// The following concept ensures a specialization of B is deduced.
template <class> class BB;
template <class V> class BB<B<V>> { }
template <class T> concept deduces_B = requires { sizeof(BB<T>); };

// The guides for B derived from the above f1' and f2' for A are as follows:
template<std::integral W>
  requires deduces_A<C<W *, W *>> && deduces_B<C<W *, W *>>
  auto f1_prime_for_B(W *, W *) -> C<W *, W *>;

template<std::integral W, class U>
  requires deduces_A<C<W *, std::type_identity_t<U>>> &&
    deduces_B<C<W *, std::type_identity_t<U>>> &&
  auto f2_prime_for_B(W *, U) -> C<W *, std::type_identity_t<U>>;
— end example

12.4.3 Viable functions

1 From the set of candidate functions constructed for a given context (12.4.2), a set of viable functions is chosen,
   from which the best function will be selected by comparing argument conversion sequences and associated constraints
   (13.5.3) for the best fit (12.4.4). The selection of viable functions considers associated constraints,
   if any, and relationships between arguments and function parameters other than the ranking of conversion sequences.

2 First, to be a viable function, a candidate function shall have enough parameters to agree in number with
   the arguments in the list.
   
   (2.1) — If there are m arguments in the list, all candidate functions having exactly m parameters are viable.
   
   (2.2) — A candidate function having fewer than m parameters is viable only if it has an ellipsis in its parameter
          list (9.3.4.6). For the purposes of overload resolution, any argument for which there is no corresponding
          parameter is considered to “match the ellipsis” (12.4.4.2.4).
   
   (2.3) — A candidate function having more than m parameters is viable only if all parameters following the
          m-th have default arguments (9.3.4.7). For the purposes of overload resolution, the parameter list is
          truncated on the right, so that there are exactly m parameters.

3 Second, for a function to be viable, if it has associated constraints (13.5.3), those constraints shall be
   satisfied (13.5.2).

4 Third, for F to be a viable function, there shall exist for each argument an implicit conversion sequence (12.4.4.2)
   that converts that argument to the corresponding parameter of F. If the parameter has reference type, the
   implicit conversion sequence includes the operation of binding the reference, and the fact that an lvalue
   reference to non-const cannot be bound to an rvalue and that an rvalue reference cannot be bound to an
   lvalue can affect the viability of the function (see 12.4.4.2.5).

12.4.4 Best viable function

12.4.4.1 General

Define ICSi(F) as follows:

   (1.1) — If F is a static member function, ICSi(F) is defined such that ICSi(F) is neither better nor worse than
          ICSi(G) for any function G, and, symmetrically, ICSi(G) is neither better nor worse than ICSi(F),127
          otherwise,

   (1.2) — let ICSi(f) denote the implicit conversion sequence that converts the i-th argument in the list to the
          type of the i-th parameter of viable function F. 12.4.4.2 defines the implicit conversion sequences and

127 If a function is a static member function, this definition means that the first argument, the implied object argument, has no
   effect in the determination of whether the function is better or worse than any other function.
12.4.4.3 defines what it means for one implicit conversion sequence to be a better conversion sequence or worse conversion sequence than another.

Given these definitions, a viable function \( F_1 \) is defined to be a better function than another viable function \( F_2 \) if for all arguments \( i \), ICS\(_i\)(\( F_1 \)) is not a worse conversion sequence than ICS\(_i\)(\( F_2 \)), and then

1. for some argument \( j \), ICS\(_j\)(\( F_1 \)) is a better conversion sequence than ICS\(_j\)(\( F_2 \)), or, if not that,
2. the context is an initialization by user-defined conversion (see 9.4, 12.4.2.6, and 12.4.2.7) and the standard conversion sequence from the return type of \( F_1 \) to the destination type (i.e., the type of the entity being initialized) is a better conversion sequence than the standard conversion sequence from the return type of \( F_2 \) to the destination type.

**Example 1:**
```cpp
struct A {
    A();
    operator int();
    operator double();
} a;
int i = a;  // a.operator int() followed by no conversion is better than
            // a.operator double() followed by a conversion to int
float x = a;  // ambiguous: both possibilities require conversions,
              // and neither is better than the other
```

or, if not that,

1. the context is an initialization by conversion function for direct reference binding (12.4.2.7) of a reference to function type, the return type of \( F_1 \) is the same kind of reference (lvalue or rvalue) as the reference being initialized, and the return type of \( F_2 \) is not

**Example 2:**
```cpp
template <class T> struct A {
    operator T&();  // #1
    operator T&&();  // #2
};
typedef int Fn();
A<Fn> a;
Fn& lf = a;  // calls #1
Fn&& rf = a;  // calls #2
```

or, if not that,

1. \( F_1 \) is not a function template specialization and \( F_2 \) is a function template specialization, or, if not that,
2. \( F_1 \) and \( F_2 \) are function template specializations, and the function template for \( F_1 \) is more specialized than the template for \( F_2 \) according to the partial ordering rules described in 13.7.7.3, or, if not that,
3. \( F_1 \) and \( F_2 \) are non-template functions with the same parameter-type-lists, and \( F_1 \) is more constrained than \( F_2 \) according to the partial ordering of constraints described in 13.5.5, or if not that,
4. \( F_1 \) is a constructor for a class \( D \), \( F_2 \) is a constructor for a base class \( B \) of \( D \), and for all arguments the corresponding parameters of \( F_1 \) and \( F_2 \) have the same type

**Example 3:**
```cpp
struct A {
    A(int = 0);
};

struct B: A {
    using A::A;
    B();
};

int main() {
    B b;  // OK, B::B()
}
```
— end example]

or, if not that,

(2.8) — F2 is a rewritten candidate (12.4.2.3) and F1 is not

[Example 4:
  struct S {
    friend auto operator<=>(const S&, const S&) = default;  // #1
    friend bool operator<(const S&, const S&);             // #2
  };
  bool b = S() < S();                                       // calls #2
— end example]

or, if not that,

(2.9) — F1 and F2 are rewritten candidates, and F2 is a synthesized candidate with reversed order of parameters and F1 is not

[Example 5:
  struct S {
    friend std::weak_ordering operator<=>(const S&, int);    // #1
    friend std::weak_ordering operator<=>(int, const S&);    // #2
  };
  bool b = 1 < S();                                         // calls #2
— end example]

or, if not that

(2.10) — F1 is generated from a deduction-guide (12.4.2.9) and F2 is not, or, if not that,

(2.11) — F1 is the copy deduction candidate (12.4.2.9) and F2 is not, or, if not that,

(2.12) — F1 is generated from a non-template constructor and F2 is generated from a constructor template.

[Example 6:
  template <class T> struct A {
    using value_type = T;
    A(value_type); // #1
    A(const A&);  // #2
    A(T, T, int); // #3
    template<class U>
    A(int, T, U); // #4
    // #5 is the copy deduction candidate, A(A)
  };

  A x(1, 2, 3); // uses #3, generated from a non-template constructor

  template <class T>
  A(T) -> A<T>; // #6, less specialized than #5

  A a(42);       // uses #6 to deduce A<int> and #1 to initialize
  A b = a;       // uses #5 to deduce A<int> and #2 to initialize

  template <class T>
  A(A<T>) -> A<A<T>>; // #7, as specialized as #5

  A b2 = a;      // uses #7 to deduce A<A<int>> and #1 to initialize
— end example]

3 If there is exactly one viable function that is a better function than all other viable functions, then it is the one selected by overload resolution; otherwise the call is ill-formed.\textsuperscript{128}

\textsuperscript{128} The algorithm for selecting the best viable function is linear in the number of viable functions. Run a simple tournament to find a function W that is not worse than any opponent it faced. Although another function F that W did not face might be at least as good as W, F cannot be the best function because at some point in the tournament F encountered another function G such that F was not better than G. Hence, either W is the best function or there is no best function. So, make a second pass over the viable functions to verify that W is better than all other functions.
[Example 7:]

```c
void Fcn(const int*, short);
void Fcn(int*, int);

int i;
short s = 0;

void f() {
    Fcn(&i, s); // is ambiguous because &i → int* is better than &i → const int*
    // but s → short is also better than s → int
    Fcn(&i, 1L); // calls Fcn(int*, int), because &i → int* is better than &i → const int*
    // and 1L → short and 1L → int are indistinguishable
    Fcn(&i, 'c'); // calls Fcn(int*, int), because &i → int* is better than &i → const int*
    // and c → int is better than c → short
}
```

—end example—

4 If the best viable function resolves to a function for which multiple declarations were found, and if at least two of these declarations — or the declarations they refer to in the case of using-declarations — specify a default argument that made the function viable, the program is ill-formed.

[Example 8:]

```c
namespace A {
    extern "C" void f(int = 5);
}
namespace B {
    extern "C" void f(int = 5);
}
using A::f;
using B::f;

void use() {
    f(3); // OK, default argument was not used for viability
    f(); // error: found default argument twice
}
```

—end example—

12.4.4.2 Implicit conversion sequences

12.4.4.2.1 General

1 An implicit conversion sequence is a sequence of conversions used to convert an argument in a function call to the type of the corresponding parameter of the function being called. The sequence of conversions is an implicit conversion as defined in 7.3, which means it is governed by the rules for initialization of an object or reference by a single expression (9.4, 9.4.4).

2 Implicit conversion sequences are concerned only with the type, cv-qualification, and value category of the argument and how these are converted to match the corresponding properties of the parameter.

[Note 1: Other properties, such as the lifetime, storage class, alignment, accessibility of the argument, whether the argument is a bit-field, and whether a function is deleted (9.5.3), are ignored. So, although an implicit conversion sequence can be defined for a given argument-parameter pair, the conversion from the argument to the parameter might still be ill-formed in the final analysis. — end note]

3 A well-formed implicit conversion sequence is one of the following forms:

1. a standard conversion sequence (12.4.4.2.2),
2. a user-defined conversion sequence (12.4.4.2.3), or
3. an ellipsis conversion sequence (12.4.4.2.4).

4 However, if the target is

1. the first parameter of a constructor or
— the implicit object parameter of a user-defined conversion function
and the constructor or user-defined conversion function is a candidate by
— 12.4.2.4, when the argument is the temporary in the second step of a class copy-initialization,
— 12.4.2.5, 12.4.2.6, or 12.4.2.7 (in all cases), or
— the second phase of 12.4.2.8 when the initializer list has exactly one element that is itself an initializer
list, and the target is the first parameter of a constructor of class \textit{X}, and the conversion is to \textit{X} or
reference to \textit{cv X},

user-defined conversion sequences are not considered.

\[\text{Note 2: These rules prevent more than one user-defined conversion from being applied during overload resolution, thereby avoiding infinite recursion. — end note}\]

\[\text{Example 1:}\]
\begin{verbatim}
struct Y { Y(int); };  
struct A { operator int() { };  
    Y y1 = A(); // error: A::operator int() is not a candidate

struct X { X(); };  
struct B { operator X() { };  
    X x{{b}}; // error: B::operator X() is not a candidate
\end{verbatim}
— end example]

For the case where the parameter type is a reference, see 12.4.4.2.5.

5 When the parameter type is not a reference, the implicit conversion sequence models a copy-initialization of
the parameter from the argument expression. The implicit conversion sequence is the one required to convert
the argument expression to a prvalue of the type of the parameter.

\[\text{Note 3: When the parameter has a class type, this is a conceptual conversion defined for the purposes of Clause 12; the actual initializtion is defined in terms of constructors and is not a conversion. — end note}\]

Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a
conversion.

\[\text{Example 2: A parameter of type } \textit{A} \text{ can be initialized from an argument of type } \textit{const A}. \text{ The implicit conversion sequence for that case is the identity sequence; it contains no “conversion” from } \textit{const A} \text{ to } \textit{A}. — end example]\n
When the parameter has a class type and the argument expression has the same type, the implicit conversion
sequence is an identity conversion. When the parameter has a class type and the argument expression has a
derived class type, the implicit conversion sequence is a derived-to-base conversion from the derived class to
the base class.

\[\text{Note 4: There is no such standard conversion; this derived-to-base conversion exists only in the description of implicit conversion sequences. — end note}\]

A derived-to-base conversion has Conversion rank (12.4.4.2.2).

In all contexts, when converting to the implicit object parameter or when converting to the left operand of
an assignment operation only standard conversion sequences are allowed.

If no conversions are required to match an argument to a parameter type, the implicit conversion sequence is
the standard conversion sequence consisting of the identity conversion (12.4.4.2.2).

If no sequence of conversions can be found to convert an argument to a parameter type, an implicit conversion
sequence cannot be formed.

If there are multiple well-formed implicit conversion sequences converting the argument to the parameter
type, the implicit conversion sequence associated with the parameter is defined to be the unique conversion
sequence designated the ambiguous conversion sequence. For the purpose of ranking implicit conversion
sequences as described in 12.4.4.3, the ambiguous conversion sequence is treated as a user-defined conversion
sequence that is indistinguishable from any other user-defined conversion sequence.

\[\text{Note 5: This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters.}\]

\[\text{Example 3}:\]

\begin{verbatim}
class B;
\end{verbatim}

\[\text{§ 12.4.4.2.1} \quad 344\]
class A { A (B&); };  
class B { operator A () ; };  
class C { C (B&); };  
void f(A) { }  
void f(C) { }  
B b;  
f(b);  // error: ambiguous because there is a conversion b → C (via constructor)  
// and an (ambiguous) conversion b → A (via constructor or conversion function)  
void f(B) { }  
f(b);  // OK, unambiguous  
—end example]  
—end note]  

If a function that uses the ambiguous conversion sequence is selected as the best viable function, the call will be ill-formed because the conversion of one of the arguments in the call is ambiguous.

11 The three forms of implicit conversion sequences mentioned above are defined in the following subclauses.

12.4.4.2.2 Standard conversion sequences

Table 16 summarizes the conversions defined in 7.3 and partitions them into four disjoint categories: Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion.

[Note 1: These categories are orthogonal with respect to value category, cv-qualification, and data representation: the Lvalue Transformations do not change the cv-qualification or data representation of the type; the Qualification Adjustments do not change the value category or data representation of the type; and the Promotions and Conversions do not change the value category or cv-qualification of the type. —end note]

[Note 2: As described in 7.3, a standard conversion sequence either is the Identity conversion by itself (that is, no conversion) or consists of one to three conversions from the other four categories. If there are two or more conversions in the sequence, the conversions are applied in the canonical order: Lvalue Transformation, Promotion or Conversion, Qualification Adjustment. —end note]

Each conversion in Table 16 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (12.4.4.3). The rank of a conversion sequence is determined by considering the rank of each conversion in the sequence and the rank of any reference binding (12.4.4.2.5). If any of those has Conversion rank, the sequence has Conversion rank; otherwise, if any of those has Promotion rank, the sequence has Promotion rank; otherwise, the sequence has Exact Match rank.

Table 16: Conversions

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Category</th>
<th>Rank</th>
<th>Subclause</th>
</tr>
</thead>
<tbody>
<tr>
<td>No conversions required</td>
<td>Identity</td>
<td></td>
<td>7.3.2</td>
</tr>
<tr>
<td>Lvalue-to-rvalue conversion</td>
<td>Lvalue Transformation</td>
<td>Exact Match</td>
<td>7.3.3</td>
</tr>
<tr>
<td>Array-to-pointer conversion</td>
<td>Qualification Adjustment</td>
<td></td>
<td>7.3.4</td>
</tr>
<tr>
<td>Function-to-pointer conversion</td>
<td></td>
<td></td>
<td>7.3.6</td>
</tr>
<tr>
<td>Qualification conversions</td>
<td></td>
<td></td>
<td>7.3.14</td>
</tr>
<tr>
<td>Function pointer conversion</td>
<td>Promotion</td>
<td>Promotion</td>
<td>7.3.7</td>
</tr>
<tr>
<td>Integral promotions</td>
<td></td>
<td></td>
<td>7.3.8</td>
</tr>
<tr>
<td>Floating-point promotion</td>
<td></td>
<td></td>
<td>7.3.9</td>
</tr>
<tr>
<td>Integral conversions</td>
<td>Conversion</td>
<td></td>
<td>7.3.10</td>
</tr>
<tr>
<td>Floating-point conversions</td>
<td></td>
<td></td>
<td>7.3.11</td>
</tr>
<tr>
<td>Floating-integral conversions</td>
<td></td>
<td></td>
<td>7.3.12</td>
</tr>
<tr>
<td>Pointer conversions</td>
<td></td>
<td></td>
<td>7.3.13</td>
</tr>
<tr>
<td>Pointer-to-member conversions</td>
<td></td>
<td></td>
<td>7.3.14</td>
</tr>
<tr>
<td>Boolean conversions</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

12.4.4.2.3 User-defined conversion sequences

A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (11.4.8) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (11.4.8.2), the initial standard conversion sequence converts the source type to the type of the first parameter of that constructor. If the user-defined conversion is specified by a conversion
function (11.4.8.3), the initial standard conversion sequence converts the source type to the type of the implicit object parameter of that conversion function.

2 The second standard conversion sequence converts the result of the user-defined conversion to the target type for the sequence; any reference binding is included in the second standard conversion sequence. Since an implicit conversion sequence is an initialization, the special rules for initialization by user-defined conversion apply when selecting the best user-defined conversion for a user-defined conversion sequence (see 12.4.4 and 12.4.4.2).

3 If the user-defined conversion is specified by a specialization of a conversion function template, the second standard conversion sequence shall have exact match rank.

4 A conversion of an expression of class type to the same class type is given Exact Match rank, and a conversion of an expression of class type to a base class of that type is given Conversion rank, in spite of the fact that a constructor (i.e., a user-defined conversion function) is called for those cases.

12.4.4.2.4 Ellipsis conversion sequences

1 An ellipsis conversion sequence occurs when an argument in a function call is matched with the ellipsis parameter specification of the function called (see 7.6.1.3).

12.4.4.2.5 Reference binding

1 When a parameter of reference type binds directly (9.4.4) to an argument expression, the implicit conversion sequence is the identity conversion, unless the argument expression has a type that is a derived class of the parameter type, in which case the implicit conversion sequence is a derived-to-base conversion (12.4.4.2). [Example 1:

```
struct A {}
struct B : public A {} b;
int f(A&);
int f(B&);
int i = f(b);       // calls f(B&), an exact match, rather than f(A&), a conversion
```
—end example] If the parameter binds directly to the result of applying a conversion function to the argument expression, the implicit conversion sequence is a user-defined conversion sequence (12.4.4.2.3), with the second standard conversion sequence either an identity conversion or, if the conversion function returns an entity of a type that is a derived class of the parameter type, a derived-to-base conversion.

2 When a parameter of reference type is not bound directly to an argument expression, the conversion sequence is the one required to convert the argument expression to the referenced type according to 12.4.4.2. Conceptually, this conversion sequence corresponds to copy-initializing a temporary of the referenced type with the argument expression. Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion.

3 Except for an implicit object parameter, for which see 12.4.2, an explicit conversion sequence cannot be formed if it requires binding an lvalue reference other than a reference to a non-volatile const type to an rvalue or binding an rvalue reference to an lvalue other than a function lvalue. [Note 1: This means, for example, that a candidate function cannot be a viable function if it has a non-const lvalue reference parameter (other than the implicit object parameter) and the corresponding argument would require a temporary to be created to initialize the lvalue reference (see 9.4.4). —end note]

4 Other restrictions on binding a reference to a particular argument that are not based on the types of the reference and the argument do not affect the formation of an implicit conversion sequence, however. [Example 2: A function with an “lvalue reference to int” parameter can be a viable candidate even if the corresponding argument is an int bit-field. The formation of implicit conversion sequences treats the int bit-field as an int lvalue and finds an exact match with the parameter. If the function is selected by overload resolution, the call will nonetheless be ill-formed because of the prohibition on binding a non-const lvalue reference to a bit-field (9.4.4). —end example]

12.4.4.2.6 List-initialization sequence

1 When an argument is an initializer list (9.4.5), it is not an expression and special rules apply for converting it to a parameter type.

2 If the initializer list is a designated-initializer-list, a conversion is only possible if the parameter has an aggregate type that can be initialized from the initializer list according to the rules for aggregate initialization (9.4.2),
in which case the implicit conversion sequence is a user-defined conversion sequence whose second standard conversion sequence is an identity conversion.

[Note 1: Aggregate initialization does not require that the members are declared in designation order. If, after overload resolution, the order does not match for the selected overload, the initialization of the parameter will be ill-formed (9.4.5).

[Example 1:

```c
struct A { int x, y; }
struct B { int y, x; }
void f(A a, int); // #1
void f(B b, ...); // #2
void g(A a); // #3
void g(B b); // #4
void h() {
f( {.x = 1, .y = 2}, 0); // OK; calls #1
f( {.y = 2, .x = 1}, 0); // error: selects #1, initialization of a fails
    // due to non-matching member order (9.4.5)
g( {.x = 1, .y = 2}); // error: ambiguous between #3 and #4
}
—end example
—end note]

3 Otherwise, if the parameter type is an aggregate class X and the initializer list has a single element of type cv U, where U is X or a class derived from X, the implicit conversion sequence is the one required to convert the element to the parameter type.

4 Otherwise, if the parameter type is a character array and the initializer list has a single element that is an appropriately-typed string-literal (9.4.3), the implicit conversion sequence is the identity conversion.

5 Otherwise, if the parameter type is `std::initializer_list<X>` and all the elements of the initializer list can be implicitly converted to X, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to X, or if the initializer list has no elements, the identity conversion. This conversion can be a user-defined conversion even in the context of a call to an initializer-list constructor.

[Example 2:

```c
void f( std::initializer_list<int> );
f( {} ); // OK: f(initializer_list<int>) identity conversion
f( {1,2,3} ); // OK: f(initializer_list<int>) identity conversion
f( {"a","b"} ); // OK: f(initializer_list<int>) integral promotion
f( {1.0} ); // error: narrowing

struct A {
    A( std::initializer_list<double> ); // #1
    A( std::initializer_list<complex<double>> ); // #2
    A( std::initializer_list<std::string> ); // #3
};
A a( 1,0.2,0 ); // OK, uses #1

void g(A);
g( { "foo", "bar" } ); // OK, uses #3

typedef int IA[3];
void h(const IA&);
h( { 1, 2, 3 } ); // OK: identity conversion
—end example]

6 Otherwise, if the parameter type is “array of N X” or “array of unknown bound of X”, if there exists an implicit conversion sequence from each element of the initializer list (and from `{}` in the former case if N exceeds the number of elements in the initializer list) to X, the implicit conversion sequence is the worst such implicit conversion sequence.

7 Otherwise, if the parameter is a non-aggregate class X and overload resolution per 12.4.2.8 chooses a single best constructor C of X to perform the initialization of an object of type X from the argument initializer list:

129) Since there are no parameters of array type, this will only occur as the referenced type of a reference parameter.
— If C is not an initializer-list constructor and the initializer list has a single element of type \(cv\ U\), where
U is X or a class derived from X, the implicit conversion sequence has Exact Match rank if U is X, or
Conversion rank if U is derived from X.

— Otherwise, the implicit conversion sequence is a user-defined conversion sequence with the second
standard conversion sequence an identity conversion.

If multiple constructors are viable but none is better than the others, the implicit conversion sequence is
the ambiguous conversion sequence. User-defined conversions are allowed for conversion of the initializer list
elements to the constructor parameter types except as noted in 12.4.4.2.

Example 3:

```cpp
struct A {
    A(std::initializer_list<int>);
};
void f(A);
f({'a', 'b'}); // OK: f(A(std::initializer_list<int>)) user-defined conversion

struct B {
    B(int, double);
};
void g(B);
g({'a', 'b'}); // OK: g(B(int, double)) user-defined conversion
g({1.0, 1.0}); // error: narrowing

void f(B);
f({'a', 'b'}); // error: ambiguous f(A) or f(B)

struct C {
    C(std::string);
};
void h(C);
h("foo"); // OK: h(C(std::string("foo")))

struct D {
    D(A, C);
};
void i(D);
i({{1,2}, {"bar"}}); // OK: i(D(A(std::initializer_list<int>{1,2}), C(std::string("bar"))))
```

8 Otherwise, if the parameter has an aggregate type which can be initialized from the initializer list according
to the rules for aggregate initialization (9.4.2), the implicit conversion sequence is a user-defined conversion
sequence with the second standard conversion sequence an identity conversion.

Example 4:

```cpp
struct A {
    int m1;
    double m2;
};

void f(A);
f({'a', 'b'}); // OK: f(A(int,double)) user-defined conversion
f({1.0}); // error: narrowing
```

9 Otherwise, if the parameter is a reference, see 12.4.4.2.5.

Note 2: The rules in this subclause will apply for initializing the underlying temporary for the reference. — end note

Example 5:

```cpp
struct A {
    int m1;
    double m2;
};
```
© ISO/IEC

void f(const &a);
f( {'a', 'b'} );  // OK: f(A(int,double)) user-defined conversion
f( {1.0} );  // error: narrowing

g(const double &);
g(1);  // same conversion as int to double
— end example]

10 Otherwise, if the parameter type is not a class:

(10.1) — if the initializer list has one element that is not itself an initializer list, the implicit conversion sequence
is the one required to convert the element to the parameter type;

[Example 6:
void f(int);
f( {'a'} );  // OK: same conversion as char to int
f( {1.0} );  // error: narrowing
— end example]

(10.2) — if the initializer list has no elements, the implicit conversion sequence is the identity conversion.

[Example 7:
void f(int);
f( () );  // OK: identity conversion
— end example]

11 In all cases other than those enumerated above, no conversion is possible.

12.4.4.3 Ranking implicit conversion sequences

This subclause defines a partial ordering of implicit conversion sequences based on the relationships better conversion sequence and better conversion. If an implicit conversion sequence S1 is defined by these rules to be a better conversion sequence than S2, then it is also the case that S2 is a worse conversion sequence than S1. If conversion sequence S1 is neither better than nor worse than conversion sequence S2, S1 and S2 are said to be indistinguishable conversion sequences.

2 When comparing the basic forms of implicit conversion sequences (as defined in 12.4.4.2)

(2.1) — a standard conversion sequence (12.4.4.2.2) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and

(2.2) — a user-defined conversion sequence (12.4.4.2.3) is a better conversion sequence than an ellipsis conversion sequence (12.4.4.2.4).

3 Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:

(3.1) — List-initialization sequence L1 is a better conversion sequence than list-initialization sequence L2 if

(3.1.1) — L1 converts to std::initializer_list<X> for some X and L2 does not, or, if not that,

(3.1.2) — L1 and L2 convert to arrays of the same element type, and either the number of elements n1 initialized by L1 is less than the number of elements n2 initialized by L2, or n1 = n2 and L2 converts to an array of unknown bound and L1 does not,

even if one of the other rules in this paragraph would otherwise apply.

[Example 1:
void f1(int);
void f1(std::initializer_list<long>);  // #2
void g1() { f1({42}); }  // chooses #2

void f2(std::pair<const char*, const char*>);  // #3
void f2(std::initializer_list<std::string>);  // #4
void g2() { f2({"foo","bar"}); }  // chooses #4
— end example]

[Example 2:
void f(int (&&)[1]);  // #1

§ 12.4.4.3 349
© ISO/IEC

void f(double (&&)[ ] );  // #2
void f(int (&&)[2]);     // #3

f( {1} );                  // Calls #1: Better than #2 due to conversion, better than #3 due to bounds
f( {1.0} );                // Calls #2: Identity conversion is better than floating-integral conversion
f( {1.0, 2.0} );          // Calls #2: Identity conversion is better than floating-integral conversion
f( {1, 2} );              // Calls #3: Converting to array of known bound is better than to unknown bound,
                          // and an identity conversion is better than floating-integral conversion

— end example —

— Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if

(3.2.1) — S1 is a proper subsequence of S2 (comparing the conversion sequences in the canonical form defined by 12.4.4.2.2, excluding any Lvalue Transformation; the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence) or, if not that,

(3.2.2) — the rank of S1 is better than the rank of S2, or S1 and S2 have the same rank and are distinguishable by the rules in the paragraph below, or, if not that,

(3.2.3) — S1 and S2 include reference bindings (9.4.4) and neither refers to an implicit object parameter of a non-static member function declared without a ref-qualifier, and S1 binds an rvalue reference to an rvalue and S2 binds an lvalue reference

[Example 3:

```c
int i;
int f1();
int&& f2();
int g(const int&);
int g(const int&&);
int j = g(i);  // calls g(const int&)
int k = g(f1()); // calls g(const int&&)
int l = g(f2()); // calls g(const int&&)
```

struct A {
    A& operator<<(int);
    void p() &;
    void p() &&;
};
A& operator<<((A&&, char);
A() << 1;               // calls A::operator<<(int)
A() << 'c';            // calls A::operator<<((A&, char)
A a;
a << 1;                // calls A::operator<<(int)
a << 'c';              // calls A::operator<<(int)
A().p();               // calls A::p()&
a.p();                // calls A::p()&&

— end example]

or, if not that,

(3.2.4) — S1 and S2 include reference bindings (9.4.4) and S1 binds an lvalue reference to a function lvalue and S2 binds an rvalue reference to a function lvalue

[Example 4:

```c
int f(void(&)());     // #1
int f(void(&&)());    // #2
void g();
int i1 = f(g);        // calls #1

— end example]

or, if not that,

(3.2.5) — S1 and S2 differ only in their qualification conversion (7.3.6) and yield similar types T1 and T2, respectively, where T1 can be converted to T2 by a qualification conversion.

[Example 5:

```c

§ 12.4.4.3

350

N4868

© ISO/IEC
int f(const volatile int *);
int f(const int *);
int i;
int j = f(&i);  // calls f(const int*)
— end example]

or, if not that,
(3.2.6)
— S1 and S2 include reference bindings (9.4.4), and the types to which the references refer are the
same type except for top-level cv-qualifiers, and the type to which the reference initialized by S2
refers is more cv-qualified than the type to which the reference initialized by S1 refers.

[Example 6:

int i;
int j = f(i);  // calls f(int &)
int k = g(i);  // ambiguous

struct X {
    void f() const;
    void f();
};
void g(const X & a, X b) {
    a.f();  // calls X::f() const
    b.f();  // calls X::f()
}
— end example]

(3.3)
— User-defined conversion sequence U1 is a better conversion sequence than another user-defined conversion
sequence U2 if they contain the same user-defined conversion function or constructor or they initialize
the same class in an aggregate initialization and in either case the second standard conversion sequence
of U1 is better than the second standard conversion sequence of U2.

[Example 7:

struct A {
    operator short();
} a;
int f(int);
int f(float);
int i = f(a);  // calls f(int), because short → int is
              // better than short → float.
— end example]

4 Standard conversion sequences are ordered by their ranks: an Exact Match is a better conversion than a
Promotion, which is a better conversion than a Conversion. Two conversion sequences with the same rank
are indistinguishable unless one of the following rules applies:

(4.1)
— A conversion that does not convert a pointer or a pointer to member to bool is better than one that
does.
(4.2)
— A conversion that promotes an enumeration whose underlying type is fixed to its underlying type is
better than one that promotes to the promoted underlying type, if the two are different.
(4.3)
— If class B is derived directly or indirectly from class A, conversion of B* to A* is better than conversion
of B* to void*, and conversion of A* to void* is better than conversion of B* to void*.
(4.4)
— If class B is derived directly or indirectly from class A and class C is derived directly or indirectly from B,
(4.4.1)
— conversion of C* to B* is better than conversion of C* to A*;

[Example 8:

struct A {};
struct B : public A {};

§ 12.4.4.3
struct C : public B {}
C* pc;
int f(A*);
int f(B*);
in i = f(pc);        // calls f(B*)

- binding of an expression of type C to a reference to type B is better than binding an expression of
type C to a reference to type A,

— conversion of A::* to B::* is better than conversion of A::* to C::*,

— conversion of C to B is better than conversion of C to A,

— binding of an expression of type B to a reference to type A is better than binding an expression of
type C to a reference to type A,

— conversion of B::* to C::* is better than conversion of A::* to C::*; and

— conversion of B to A is better than conversion of C to A.

[Note 1: Compared conversion sequences will have different source types only in the context of comparing the
second standard conversion sequence of an initialization by user-defined conversion (see 12.4.4); in all other
contexts, the source types will be the same and the target types will be different. — end note]

12.5 Address of overloaded function

A use of a function name without arguments is resolved to a function, a pointer to function, or a pointer
to member function for a specific function that is chosen from a set of selected functions determined based on
the target type required in the context (if any), as described below. The target can be

1. an object or reference being initialized (9.4, 9.4.4, 9.4.5),
2. the left side of an assignment (7.6.19),
3. a parameter of a function (7.6.1.3),
4. a parameter of a user-defined operator (12.6),
5. the return value of a function, operator function, or conversion (8.7.4),
6. an explicit type conversion (7.6.1.4, 7.6.1.9, 7.6.3), or
7. a non-type template-parameter (13.4.3).

The function name can be preceded by the & operator.

[Note 1: Any redundant set of parentheses surrounding the function name is ignored (7.5.3). — end note]

If there is no target, all non-template functions named are selected. Otherwise, a non-template function with
type F is selected for the function type FT of the target type if F (after possibly applying the function pointer
conversion (7.3.14)) is identical to FT.

[Note 2: That is, the class of which the function is a member is ignored when matching a pointer-to-member-function
type. — end note]

For each function template designated by the name, template argument deduction is done (13.10.3.3), and if
the argument deduction succeeds, the resulting template argument list is used to generate a single function
template specialization, which is added to the set of selected functions considered.

[Note 3: As described in 13.10.2, if deduction fails and the function template name is followed by an explicit template
argument list, the template-id is then examined to see whether it identifies a single function template specialization. If
it does, the template-id is considered to be an lvalue for that function template specialization. The target type is not
used in that determination. — end note]

Non-member functions and static member functions match targets of function pointer type or reference to
function type. Non-static member functions match targets of pointer-to-member-function type. If a non-static
member function is selected, the reference to the overloaded function name is required to have the form of a
pointer to member as described in 7.6.2.2.

All functions with associated constraints that are not satisfied (13.5.3) are eliminated from the set of selected
functions. If more than one function in the set remains, all function template specializations in the set are
eliminated if the set also contains a function that is not a function template specialization. Any given non-template function \( F_0 \) is eliminated if the set contains a second non-template function that is more constrained than \( F_0 \) according to the partial ordering rules of 13.5.5. Any given function template specialization \( F_1 \) is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of \( F_1 \) according to the partial ordering rules of 13.7.7.3. After such eliminations, if any, there shall remain exactly one selected function.

6 [Example 1:

```c
int f(double);
int f(int);
int (*pf)(double) = &f; // selects f(double)
int (*pfi)(int) = &f; // selects f(int)
int (*pfe)(...) = &f; // error: type mismatch
int (&rfi)(int) = f; // selects f(int)
int (&rfd)(double) = f; // selects f(double)
void g() {
    (int (*)(int))&f; // cast expression as selector
}
```

The initialization of \( pfe \) is ill-formed because no \( f() \) with type \( \text{int}(...) \) has been declared, and not because of any ambiguity. For another example,

```c
struct X {
    int f(int);
    static int f(long);
};
int (X::*p1)(int) = &X::f; // OK
int (*p2)(int) = &X::f; // error: mismatch
int (X::*p4)(long) = &X::f; // error: mismatch
int (X::*p5)(int) = &X::f; // error: wrong syntax for // pointer to member
int (*p6)(long) = &X::f; // OK
```

7 [Note 4: If \( f() \) and \( g() \) are both overloaded functions, the Cartesian product of possibilities is considered to resolve \( f(&g) \), or the equivalent expression \( f(g) \). — end note]

8 [Note 5: Even if \( B \) is a public base of \( D \), we have

```c
D* f();
B* (*p1)() = &f; // error
```

—end note]

12.6 Overloaded operators [over.oper]

12.6.1 General [over.oper.general]

A function declaration having one of the following operator-function-ids as its name declares an operator function. A function template declaration having one of the following operator-function-ids as its name declares an operator function template. A specialization of an operator function template is also an operator function. An operator function is said to implement the operator named in its operator-function-id.
§ 12.6.2 Unary operators

A prefix unary operator function is a function named `operator@` for a prefix unary operator `@` (7.6.2.2) that is either a non-static member function (11.4.2) with no parameters or a non-member function with one parameter. For a unary-expression of the form `cast-expression`, the operator function is selected by overload resolution (12.4.2.3). If a member function is selected, the expression is interpreted as `cast-expression . operator @ ()`

Otherwise, if a non-member function is selected, the expression is interpreted as `operator @ ( cast-expression )`

[Note 1: The operators `new[]`, `delete[]`, `()`, and `[]` are formed from more than one token. The latter two operators are function call (7.6.1.3) and subscripting (7.6.1.2). — end note]

2 Both the unary and binary forms of

```
+ - * &
```

can be overloaded.

3 [Note 2: The following operators cannot be overloaded:

```
. * :: ?:
```

nor can the preprocessing symbols `#` (15.6.3) and `##` (15.6.4). — end note]

4 Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (12.6.2 – 12.6.7). They can be explicitly called, however, using the `operator-function-id` as the name of the function in the function call syntax (7.6.1.3).

```
Example 1:
complex z = a.operator+(b);  // complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

—end example

5 The allocation and deallocation functions, `operator new`, `operator new[]`, `operator delete`, and `operator delete[]`, are described completely in 6.7.5.5. The attributes and restrictions found in the rest of 12.6 do not apply to them unless explicitly stated in 6.7.5.5.

6 The `co_await` operator is described completely in 7.6.2.4. The attributes and restrictions found in the rest of 12.6 do not apply to it unless explicitly stated in 7.6.2.4.

7 An operator function shall either be a non-static member function or be a non-member function that has at least one parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration. It is not possible to change the precedence, grouping, or number of operands of operators. The meaning of the operators `=`, `(unary) &`, and `, (comma)`, predefined for each type, can be changed for specific class types by defining operator functions that implement these operators. Likewise, the meaning of the operators `(unary) &` and `, (comma)` can be changed for specific enumeration types. Operator functions are inherited in the same manner as other base class functions.

8 An operator function shall be a prefix unary, binary, function call, subscripting, class member access, increment, or decrement operator function.

9 [Note 3: The identities among certain predefined operators applied to basic types (for example, `++a ≡ a+=1`) need not hold for operator functions. Some predefined operators, such as `+=`, require an operand to be an lvalue when applied to basic types; this is not required by operator functions. — end note]

10 An operator function cannot have default arguments (9.3.4.7), except where explicitly stated below. Operator functions cannot have more or fewer parameters than the number required for the corresponding operator, as described in the rest of 12.6.

11 Operators not mentioned explicitly in subclauses 12.6.3.2 through 12.6.7 act as ordinary unary and binary operators obeying the rules of 12.6.2 or 12.6.3.
12.6.3 Binary operators

12.6.3.1 General

A binary operator function is a function named `operator@` for a binary operator `@` that is either a non-static member function (11.4.2) with one parameter or a non-member function with two parameters. For an expression `x @ y` with subexpressions `x` and `y`, the operator function is selected by overload resolution (12.4.2.3). If a member function is selected, the expression is interpreted as

```cpp
x . operator @ ( y )
```

Otherwise, if a non-member function is selected, the expression is interpreted as

```cpp
operator @ ( x , y )
```

An equality operator function is an operator function for an equality operator (7.6.10). A relational operator function is an operator function for a relational operator (7.6.9). A three-way comparison operator function is an operator function for the three-way comparison operator (7.6.8). A comparison operator function is an equality operator function, a relational operator function, or a three-way comparison operator function.

12.6.3.2 Simple assignment

A simple assignment operator function is a binary operator function named `operator=`. A simple assignment operator function shall be a non-static member function.

1 [Note 1: Because only standard conversion sequences are considered when converting to the left operand of an assignment operation (12.4.4.2), an expression `x = y` with a subexpression `x` of class type is always interpreted as `x.operator=(y)`. — end note]

2 [Note 2: Since a copy assignment operator is implicitly declared for a class if not declared by the user (11.4.6), a base class assignment operator function is always hidden by the copy assignment operator function of the derived class. — end note]

3 [Note 3: Any assignment operator function, even the copy and move assignment operators, can be virtual. For a derived class `D` with a base class `B` for which a virtual copy/move assignment has been declared, the copy/move assignment operator in `D` does not override `B`'s virtual copy/move assignment operator.

```cpp
Example 1:

struct B {
    virtual int operator=(int);
    virtual B& operator=(const B&);
};
struct D : B {
    virtual int operator=(int);
    virtual D& operator=(const B&);
};
D dobj1;
D dobj2;
B* bptr = &dobj1;
void f() {
    bptr->operator=(99);  // calls D::operator=(int)
    *bptr = 99;           // ditto
    bptr->operator=(dobj2); // calls D::operator=(const B&)
    *bptr = dobj2;        // ditto
    dobj1 = dobj2;        // calls implicitly-declared D::operator=(const D&)
}
— end example
— end note

12.6.4 Function call

A function call operator function is a function named `operator()` that is a non-static member function with an arbitrary number of parameters. It may have default arguments. For an expression of the form

```cpp
postfix-expression ( expression-listopt )
```

where the postfix-expression is of class type, the operator function is selected by overload resolution (12.4.2.3). If a surrogate call function for a conversion function named `operator conversion-type-id` is selected, the expression is interpreted as

```cpp
postfix-expression . operator conversion-type-id () ( expression-listopt )
```
Otherwise, the expression is interpreted as

\[
\text{postfix-expression . operator () ( expression-list_opt )}
\]

### 12.6.5 Subscripting

A subscripting operator function is a function named `operator[]` that is a non-static member function with exactly one parameter. For an expression of the form

\[
\text{postfix-expression [ expr-or-braced-init-list ]}
\]

the operator function is selected by overload resolution (12.4.2.3). If a member function is selected, the expression is interpreted as

\[
\text{postfix-expression . operator [] ( expr-or-braced-init-list )}
\]

1. **Example 1:**

```cpp
struct X {
    Z operator[](std::initializer_list<int>);
};
X x;
x[{1,2,3}] = 7; // OK: meaning x.operator[](\{1,2,3\})
int a[10];
a[{1,2,3}] = 7; // error: built-in subscript operator
```

### 12.6.6 Class member access

A class member access operator function is a function named `operator->` that is a non-static member function taking no parameters. For an expression of the form

\[
\text{postfix-expression -> template_opt id-expression}
\]

the operator function is selected by overload resolution (12.4.2.3), and the expression is interpreted as

\[
( \text{postfix-expression . operator -> () } ) -> \text{template_opt id-expression}
\]

### 12.6.7 Increment and decrement

An increment operator function is a function named `operator++`. If this function is a non-static member function with no parameters, or a non-member function with one parameter, it defines the prefix increment operator `++` for objects of that type. If the function is a non-static member function with one parameter (which shall be of type `int`) or a non-member function with two parameters (the second of which shall be of type `int`), it defines the postfix increment operator `++` for objects of that type. When the postfix increment is called as a result of using the `++` operator, the `int` argument will have value zero.

1. **Example 1:**

```cpp
struct X {
    X& operator++(); // prefix ++
    X operator++(int); // postfix a++
};
struct Y {
    Y& operator++(Y&); // prefix ++b
    Y operator++(Y&, int); // postfix b++
}
void f(X a, Y b) {
    ++a; // a.operator++();
    a++; // a.operator++(0);
    ++b; // operator++(b);
    b++; // operator++(b, 0);
    a.operator++(); // explicit call: like ++a;
    a.operator++(0); // explicit call: like a++;
    operator++(b); // explicit call: like ++b;
    operator++(b, 0); // explicit call: like b++;
}
```

130) Calling `operator++` explicitly, as in expressions like `a.operator++(2)`, has no special properties: The argument to `operator++` is 2.
A decrement operator function is a function named `operator--` and is handled analogously to an increment operator function.

### 12.7 Built-in operators

The candidate operator functions that represent the built-in operators defined in 7.6 are specified in this subclause. These candidate functions participate in the operator overload resolution process as described in 12.4.2.3 and are used for no other purpose.

**Note 1:** Because built-in operators take only operands with non-class type, and operator overload resolution occurs only when an operand expression originally has class or enumeration type, operator overload resolution can resolve to a built-in operator only when an operand has a class type that has a user-defined conversion to a non-class type appropriate for the operator, or when an operand has an enumeration type that can be converted to a type appropriate for the operator. Also note that some of the candidate operator functions given in this subclause are more permissive than the built-in operators themselves. As described in 12.4.2.3, after a built-in operator is selected by overload resolution the expression is subject to the requirements for the built-in operator given in 7.6, and therefore to any additional semantic constraints given there. If there is a user-written candidate with the same name and parameter types as a built-in candidate operator function, the built-in operator function is hidden and is not included in the set of candidate functions. —end note

In this subclause, the term *promoted integral type* is used to refer to those integral types which are preserved by integral promotion (7.3.7) (including e.g. `int` and `long` but excluding e.g. `char`).

**Note 2:** In all cases where a promoted integral type is required, an operand of unscoped enumeration type will be acceptable by way of the integral promotions. —end note

In the remainder of this subclause, `vq` represents either `volatile` or no cv-qualifier.

For every pair `(T, vq)`, where `T` is an arithmetic type other than `bool`, there exist candidate operator functions of the form

- `vq T k` operator++(`vq T k`);
- `T` operator++(`vq T k`, `int`);

For every pair `(T, vq)`, where `T` is an arithmetic type other than `bool`, there exist candidate operator functions of the form

- `vq T k` operator--(`vq T k`);
- `T` operator--(`vq T k`, `int`);

For every pair `(T, vq)`, where `T` is a cv-qualified or cv-unqualified object type, there exist candidate operator functions of the form

- `T* vq k` operator++(`T* vq k`);
- `T* vq k` operator--(`T* vq k`);
- `T*` operator++(`T* vq k`, `int`);
- `T*` operator--(`T* vq k`, `int`);

For every cv-qualified or cv-unqualified object type `T`, there exist candidate operator functions of the form

- `T k` operator*(`T*`);

For every function type `T` that does not have cv-qualifiers or a `ref-qualifier`, there exist candidate operator functions of the form

- `T k` operator*(`T*`);

For every cv-qualified or cv-unqualified object type `T`, there exist candidate operator functions of the form

- `T*` operator*(`T*`);

For every floating-point or promoted integral type `T`, there exist candidate operator functions of the form

- `T` operator*(`T`);
- `T` operator*(`T`);

For every promoted integral type `T`, there exist candidate operator functions of the form

- `T` operator*(`T`);

For every quintuple `(C1, C2, T, cv1, cv2)`, where `C2` is a class type, `C1` is the same type as `C2` or is a derived class of `C2`, and `T` is an object type or a function type, there exist candidate operator functions of the form

- `cv12 T k` operator->*(`cv1 C1*`, `cv2 T C2::*`);
where \( cv2 \) is the union of \( cv1 \) and \( cv2 \). The return type is shown for exposition only; see 7.6.4 for the determination of the operator’s result type.

13 For every pair of types \( L \) and \( R \), where each of \( L \) and \( R \) is a floating-point or promoted integral type, there exist candidate operator functions of the form

\[
LR \quad \text{operator*}(L, R);
\]
\[
LR \quad \text{operator/}(L, R);
\]
\[
LR \quad \text{operator-}(L, R);
\]
\[
\text{bool} \quad \text{operator==}(L, R);
\]
\[
\text{bool} \quad \text{operator!=}(L, R);
\]
\[
\text{bool} \quad \text{operator<}(L, R);
\]
\[
\text{bool} \quad \text{operator>}(L, R);
\]
\[
\text{bool} \quad \text{operator<=(L, R)};
\]
\[
\text{bool} \quad \text{operator>=(L, R)};
\]

where \( LR \) is the result of the usual arithmetic conversions (7.4) between types \( L \) and \( R \).

14 For every integral type \( T \) there exists a candidate operator function of the form

\[
\text{std::strong_ordering} \quad \text{operator<=>}(T, T);
\]

15 For every pair of floating-point types \( L \) and \( R \), there exists a candidate operator function of the form

\[
\text{std::partial_ordering} \quad \text{operator<=>}(L, R);
\]

16 For every cv-qualified or cv-unqualified object type \( T \) there exist candidate operator functions of the form

\[
T* \quad \text{operator+}(T*, \text{std::ptrdiff_t});
\]
\[
T\& \quad \text{operator[]}(T*, \text{std::ptrdiff_t});
\]
\[
T* \quad \text{operator-}(T*, \text{std::ptrdiff_t});
\]
\[
T\& \quad \text{operator[]}(\text{std::ptrdiff_t}, T*);
\]

17 For every \( T \), where \( T \) is a pointer to object type, there exist candidate operator functions of the form

\[
\text{std::ptrdiff_t} \quad \text{operator-}(T, T);
\]

18 For every \( T \), where \( T \) is an enumeration type or a pointer type, there exist candidate operator functions of the form

\[
\text{bool} \quad \text{operator==}(T, T);
\]
\[
\text{bool} \quad \text{operator!=}(T, T);
\]
\[
\text{bool} \quad \text{operator<}(T, T);
\]
\[
\text{bool} \quad \text{operator>}(T, T);
\]
\[
\text{bool} \quad \text{operator<=(T, T)};
\]
\[
R \quad \text{operator>=(T, T)};
\]

where \( R \) is the result type specified in 7.6.8.

19 For every \( T \), where \( T \) is a pointer-to-member type or \( \text{std::nullptr_t} \), there exist candidate operator functions of the form

\[
\text{bool} \quad \text{operator==}(T, T);
\]
\[
\text{bool} \quad \text{operator!=}(T, T);
\]

20 For every pair of promoted integral types \( L \) and \( R \), there exist candidate operator functions of the form

\[
LR \quad \text{operator%}(L, R);
\]
\[
LR \quad \text{operator&}(L, R);
\]
\[
LR \quad \text{operator^}(L, R);
\]
\[
LR \quad \text{operator|}(L, R);
\]
\[
L \quad \text{operator<<}(L, R);
\]
\[
L \quad \text{operator>>(L, R)};
\]

where \( LR \) is the result of the usual arithmetic conversions (7.4) between types \( L \) and \( R \).

21 For every triple \( (L, vq, R) \), where \( L \) is an arithmetic type, and \( R \) is a floating-point or promoted integral type, there exist candidate operator functions of the form

\[
vq Lk \quad \text{operator=(vq Lk, R)};
\]
\[
vq Lk \quad \text{operator*=(vq Lk, R)};
\]
\[
vq Lk \quad \text{operator/=(vq Lk, R)};
\]
For every pair \((T, vq)\), where \(T\) is any type, there exist candidate operator functions of the form
\[
T*vq k\ 	ext{operator} = (T*vq k, T);
\]

For every pair \((T, vq)\), where \(T\) is an enumeration or pointer-to-member type, there exist candidate operator functions of the form
\[
vq T k\ 	ext{operator} = (vq T k, T);
\]

For every pair \((T, vq)\), where \(T\) is a cv-qualified or cv-unqualified object type, there exist candidate operator functions of the form
\[
T*vq k\ 	ext{operator} += (T*vq k, std::ptrdiff_t);
T*vq k\ 	ext{operator} -= (T*vq k, std::ptrdiff_t);
\]

For every triple \((L, vq, R)\), where \(L\) is an integral type, and \(R\) is a promoted integral type, there exist candidate operator functions of the form
\[
vq L k\ 	ext{operator} %= (vq L k, R);
vq L k\ 	ext{operator} <<= (vq L k, R);
vq L k\ 	ext{operator} >>= (vq L k, R);
vq L k\ 	ext{operator} &= (vq L k, R);
vq L k\ 	ext{operator} ^= (vq L k, R);
vq L k\ 	ext{operator} |= (vq L k, R);
\]

There also exist candidate operator functions of the form
\[
bool\ 	ext{operator} ! (bool);
bool\ 	ext{operator} & (bool, bool);
bool\ 	ext{operator} | (bool, bool);
\]

For every pair of types \(L\) and \(R\), where each of \(L\) and \(R\) is a floating-point or promoted integral type, there exist candidate operator functions of the form
\[
LR\ 	ext{operator} ? : (bool, L, R);
\]

where \(LR\) is the result of the usual arithmetic conversions (7.4) between types \(L\) and \(R\).

[Note 3: As with all these descriptions of candidate functions, this declaration serves only to describe the built-in operator for purposes of overload resolution. The operator “?:” cannot be overloaded. —end note]

For every type \(T\), where \(T\) is a pointer, pointer-to-member, or scoped enumeration type, there exist candidate operator functions of the form
\[
T\ 	ext{operator} ? : (bool, T, T);
\]

### 12.8 User-defined literals

**literal-operator-id:**

operator string-literal identifier
operator user-defined-string-literal

The **string-literal** or **user-defined-string-literal** in a literal-operator-id shall have no encoding-prefix and shall contain no characters other than the implicit terminating ’\0’. The ud-suffix of the user-defined-string-literal or the identifier in a literal-operator-id is called a **literal suffix identifier**. Some literal suffix identifiers are reserved for future standardization; see 16.4.5.3.6. A declaration whose literal-operator-id uses such a literal suffix identifier is ill-formed, no diagnostic required.

A declaration whose declarator-id is a literal-operator-id shall be a declaration of a namespace-scope function or function template (it could be a friend function (11.9.4)), an explicit instantiation or specialization of a function template, or a using-declaration (9.9). A function declared with a literal-operator-id is a **literal operator**. A function template declared with a literal-operator-id is a **literal operator template**.

The declaration of a literal operator shall have a parameter-declaration-clause equivalent to one of the following:

```c
const char*
unsigned long long int
long double
char
wchar_t
char8_t
char16_t
```
char32_t
const char*, std::size_t
const wchar_t*, std::size_t
const char8_t*, std::size_t
const char16_t*, std::size_t
const char32_t*, std::size_t

If a parameter has a default argument (9.3.4.7), the program is ill-formed.

4 A raw literal operator is a literal operator with a single parameter whose type is const char*.

5 A numeric literal operator template is a literal operator template whose template-parameter-list has a single template-parameter that is a non-type template parameter pack (13.7.4) with element type char. A string literal operator template is a literal operator template whose template-parameter-list comprises a single non-type template-parameter of class type. The declaration of a literal operator template shall have an empty parameter-declaration-clause and shall declare either a numeric literal operator template or a string literal operator template.

6 Literal operators and literal operator templates shall not have C language linkage.

7 [Note 1: Literal operators and literal operator templates are usually invoked implicitly through user-defined literals (5.13.8). However, except for the constraints described above, they are ordinary namespace-scope functions and function templates. In particular, they are looked up like ordinary functions and function templates and they follow the same overload resolution rules. Also, they can be declared inline or constexpr, they can have internal, module, or external linkage, they can be called explicitly, their addresses can be taken, etc. — end note]

8 [Example 1:

```c
void operator ""_km(long double); // OK
string operator "" _i18n(const char*, std::size_t); // OK
template <char...> double operator ""_u03C0(); // OK: UCN for lowercase pi
float operator "" _e(const char*); // OK
float operator ""E(const char*); // ill-formed, no diagnostic required:
  // reserved literal suffix (16.4.5.3.6, 5.13.8)
double operator""_Bq(long double); // OK: does not use the reserved identifier _Bq (5.10)
double operator""_Bq(long double); // ill-formed, no diagnostic required:
  // uses the reserved identifier _Bq (5.10)
float operator "" B(const char*); // error: non-empty string-literal
string operator "" 5X(const char*, std::size_t); // error: invalid literal suffix identifier
double operator "" _miles(double); // error: invalid parameter-declaration-clause
template <char...> int operator ""_j(const char*); // error: invalid parameter-declaration-clause
extern "O" void operator "" _m(long double); // error: C language linkage
```

— end example]
13 Templates

13.1 Preamble

A template defines a family of classes, functions, or variables, an alias for a family of types, or a concept.

```
template-declaration:
    template-head declaration
    template-head concept-definition

template-head:
    template < template-parameter-list > requires-clause_opt

template-parameter-list:
    template-parameter
    template-parameter-list , template-parameter

requires-clause:
    requires constraint-logical-or-expression

constraint-logical-or-expression:
    constraint-logical-and-expression
    constraint-logical-or-expression || constraint-logical-and-expression

constraint-logical-and-expression:
    primary-expression
    constraint-logical-and-expression && primary-expression
```

[Note 1: The > token following the template-parameter-list of a template-declaration can be the product of replacing a >> token by two consecutive > tokens (13.3). — end note]

The declaration in a template-declaration (if any) shall

1. declare or define a function, a class, or a variable, or
2. define a member function, a member class, a member enumeration, or a static data member of a class template or of a class nested within a class template, or
3. define a member template of a class or class template, or
4. be a deduction-guide, or
5. be an alias-declaration.

A template-declaration is a declaration. A declaration introduced by a template declaration of a variable is a variable template. A variable template at class scope is a static data member template.

[Example 1]
```
template<class T>
    constexpr T pi = T(3.1415926535897932385L);
template<class T>
    T circular_area(T r) {
        return pi<T> * r * r;
    }

struct matrix_constants {
    template<class T>
        using pauli = hermitian_matrix<T, 2>;
    template<class T>
        constexpr static pauli<T> sigma1 = { { 0, 1 }, { 1, 0 } };
    template<class T>
        constexpr static pauli<T> sigma2 = { { 0, -1i }, { 1i, 0 } };
    template<class T>
        constexpr static pauli<T> sigma3 = { { 1, 0 }, { 0, -1 } };
};
```

— end example]
A template-declaration can appear only as a namespace scope or class scope declaration. Its declaration shall not be an export-declaration. In a function template declaration, the last component of the declarator-id shall not be a template-id.

[Note 2: That last component can be an identifier, an operator-function-id, a conversion-function-id, or a literal-operator-id. In a class template declaration, if the class name is a simple-template-id, the declaration declares a class template partial specialization (13.7.6). — end note]

In a template-declaration, explicit specialization, or explicit instantiation the init-declarator-list in the declaration shall contain at most one declarator. When such a declaration is used to declare a class template, no declarator is permitted.

A template name has linkage (6.6). Specializations (explicit or implicit) of a template that has internal linkage are distinct from all specializations in other translation units. A template, a template explicit specialization (13.9.4), and a class template partial specialization shall not have C linkage. Use of a linkage specification other than "C" or "C++" with any of these constructs is conditionally-supported, with implementation-defined semantics. Template definitions shall obey the one-definition rule (6.3).

[Note 3: Default arguments for function templates and for member functions of class templates are considered definitions for the purpose of template instantiation (13.7) and must also obey the one-definition rule. — end note]

A class template shall not have the same name as any other template, class, function, variable, enumeration, enumerator, namespace, or type in the same scope (6.4), except as specified in 13.7.6. Except that a function template can be overloaded either by non-template functions (9.3.4.6) with the same name or by other function templates with the same name (13.10.4), a template name declared in namespace scope or in class scope shall be unique in that scope.

An entity is templated if it is

1. a template,
2. an entity defined (6.2) or created (6.7.7) in a templated entity,
3. a member of a templated entity,
4. an enumerator for an enumeration that is a templated entity, or
5. the closure type of a lambda-expression (7.5.5.2) appearing in the declaration of a templated entity.

[Note 4: A local class, a local variable, or a friend function defined in a templated entity is a templated entity. — end note]

A template-declaration is written in terms of its template parameters. The optional requires-clause following a template-parameter-list allows the specification of constraints (13.5.3) on template arguments (13.4). The requires-clause introduces the constraint-expression that results from interpreting the constraint-logical-or-expression as a constraint-expression. The constraint-logical-or-expression of a requires-clause is an unevaluated operand (7.2.3).

[Note 5: The expression in a requires-clause uses a restricted grammar to avoid ambiguities. Parentheses can be used to specify arbitrary expressions in a requires-clause.

[Example 2:]

```cpp
template<int N> requires N == sizeof new unsigned short
int f(); // error: parentheses required around == expression
```

— end example]

— end note]

A definition of a function template, member function of a class template, variable template, or static data member of a class template shall be reachable from the end of every definition domain (6.3) in which it is implicitly instantiated (13.9.2) unless the corresponding specialization is explicitly instantiated (13.9.3) in some translation unit; no diagnostic is required.

### 13.2 Template parameters

The syntax for template-parameters is:

```cpp
template-parameter:
  type-parameter
  parameter-declaration
```
type-parameter:
  type-parameter-key ...opt identifieropt
type-parameter-key identifieropt = type-id
type-constraint ...opt identifieropt
type-constraint identifieropt = type-id
template-head type-parameter-key ...opt identifieropt
template-head type-parameter-key identifieropt = id-expression

type-parameter-key:
class typename
type-constraint:
  nested-name-specifieropt concept-name
  nested-name-specifieropt concept-name < template-argument-listopt >

[Note 1: The > token following the template-parameter-list of a type-parameter can be the product of replacing a >> token by two consecutive > tokens (13.3). — end note]

2 There is no semantic difference between class and typename in a type-parameter-key. typename followed by an unqualified-id names a template type parameter. typename followed by a qualified-id denotes the type in a non-type parameter-declaration. A template-parameter of the form class identifier is a type-parameter.

[Example 1:]
class T { /* ... */ }; int i;

template<class T, T i> void f(T t) {
  T t1 = i; // template-parameters T and i
  ::T t2 = ::i; // global namespace members T and i
}

Here, the template f has a type-parameter called T, rather than an unnamed non-type template-parameter of class T.

—end example]

A storage class shall not be specified in a template-parameter declaration. Types shall not be defined in a template-parameter declaration.

3 A type-parameter whose identifier does not follow an ellipsis defines its identifier to be a typedef-name (if declared without template) or template-name (if declared with template) in the scope of the template declaration.

[Note 2: A template argument can be a class template or alias template. For example,

template<class T> class myarray { /* ... */ };

 template<class K, class V, template<class T> class C = myarray>
 class Map {
  C<K> key;
  C<V> value;
};

—end note]

4 A type-constraint Q that designates a concept C can be used to constrain a contextually-determined type or template type parameter pack T with a constraint-expression E defined as follows. If Q is of the form C<A₁, ···, Aₙ>, then let E' be C<T, A₁, ···, Aₙ>. Otherwise, let E' be C<T>. If T is not a pack, then E is E', otherwise E is (E' && ...). This constraint-expression E is called the immediately-declared constraint of Q for T. The concept designated by a type-constraint shall be a type concept (13.7.9).

5 A type-parameter that starts with a type-constraint introduces the immediately-declared constraint of the type-constraint for the parameter.

[Example 2:]
template<typename T> concept C1 = true;
template<typename... Ts> concept C2 = true;
template<typename T, typename U> concept C3 = true;

131) Since template template-parameters and template template-arguments are treated as types for descriptive purposes, the terms non-type parameter and non-type argument are used to refer to non-type, non-template parameters and arguments.
6 A non-type template-parameter shall have one of the following (possibly cv-qualified) types:

(6.1) — a structural type (see below),
(6.2) — a type that contains a placeholder type (9.2.9.6), or
(6.3) — a placeholder for a deduced class type (9.2.9.7).

The top-level cv-qualifiers on the template-parameter are ignored when determining its type.

7 A structural type is one of the following:

(7.1) — a scalar type, or
(7.2) — an lvalue reference type, or
(7.3) — a literal class type with the following properties:

(7.3.1) — all base classes and non-static data members are public and non-mutable and
(7.3.2) — the types of all base classes and non-static data members are structural types or (possibly multi-dimensional) array thereof.

8 An id-expression naming a non-type template-parameter of class type T denotes a static storage duration object of type const T, known as a template parameter object, whose value is that of the corresponding template argument after it has been converted to the type of the template-parameter. All such template parameters in the program of the same type with the same value denote the same template parameter object. A template parameter object shall have constant destruction (7.7).

[Note 3: If an id-expression names a non-type non-reference template-parameter, then it is a prvalue if it has non-class type. Otherwise, if it is of class type T, it is an lvalue and has type const T (7.5.4.2). —end note]

[Example 3]:

using X = int;
struct A {};

template<const X& x, int i, A a> void f() {
  i++;
  // error: change of template-parameter value
  &x;
  // OK
  &i;
  // error: address of non-reference template-parameter
  &a;
  // OK
  int& ri = i;
  // error: non-const reference bound to temporary
  const int& cri = i;
  // OK: const reference bound to temporary
  const A& ra = a;
  // OK: const reference bound to a template parameter object
}

—end example]

9 [Note 4: A non-type template-parameter cannot be declared to have type cv void.

[Example 4]:

template<void v> class X;  // error
template<void* pv> class Y;  // OK

—end example]

—end note]

10 A non-type template-parameter of type “array of T” or of function type T is adjusted to be of type “pointer to T”.

[Example 5]:

template<int* a> struct R { /* ... */ };
template<int b[5]> struct S { /* ... */ };
int p;
R<&p> v;  // OK
A non-type template parameter declared with a type that contains a placeholder type with a type-constraint introduces the immediately-declared constraint of the type-constraint for the invented type corresponding to the placeholder (9.3.4.6).

A default template-argument is a template-argument (13.4) specified after in a template-parameter. A default template-argument may be specified for any kind of template-parameter (type, non-type, template) that is not a template parameter pack (13.7.4). A default template-argument may be specified in a template declaration. A default template-argument shall not be specified in the template-parameter-lists of the definition of a member of a class template that appears not be specified in a friend class template declaration. If a friend function template declaration specifies a default template-argument, that declaration shall be a definition and shall be the only declaration of the function template in the translation unit.

The set of default template-arguments available for use is obtained by merging the default arguments from all prior declarations of the template in the same way default function arguments are (9.3.4.7).

If a template-parameter of a class template, variable template, or alias template has a default template-argument, each subsequent template-parameter shall either have a default template-argument supplied or be a template parameter pack. If a template-parameter of a primary class template, primary variable template, or alias template is a template parameter pack, it shall be the last template-parameter. A template parameter pack of a function template shall not be followed by another template parameter unless that template parameter can be deduced from the parameter-type-list (9.3.4.6) of the function template or has a default argument (13.10.3). A template parameter of a deduction guide template (13.7.2.3) that does not have a default argument shall be deducible from the parameter-type-list of the deduction guide template.

A template-parameter shall not be given default arguments by two different declarations in the same scope.

When parsing a default template-argument for a non-type template-parameter, the first non-nested > is taken as the end of the template-parameter-list rather than a greater-than operator.
A template-parameter of a template template-parameter is permitted to have a default template-argument. When such default arguments are specified, they apply to the template template-parameter in the scope of the template template-parameter.

[Example 10:]

```cpp
template <template <class TT = float> class T> struct A {
   inline void f();
   inline void g();
};
template <template <class TT> class T> void A<T>::f() {
   T<> t;     // error: TT has no default template argument
}
template <template <class TT = char> class T> void A<T>::g() {
   T<> t;     // OK, T<char>
}
```

—end example]

If a template-parameter is a type-parameter with an ellipsis prior to its optional identifier or is a parameter-declaration that declares a pack (9.3.4.6), then the template-parameter is a template parameter pack (13.7.4). A template parameter pack that is a parameter-declaration whose type contains one or more unexpanded packs is a pack expansion. Similarly, a template parameter pack that is a type-parameter with a template-parameter-list containing one or more unexpanded packs is a pack expansion. A type parameter pack with a type-constraint that contains an unexpanded parameter pack is a pack expansion. A template parameter pack that is a pack expansion shall not expand a template parameter pack declared in the same template-parameter-list.

[Example 11:]

```cpp
template <class... Types> // Types is a template type parameter pack
class Tuple;          // but not a pack expansion

template <class T, int... Dims> // Dims is a non-type template parameter pack
   struct multi_array;          // but not a pack expansion

template <class... T> // Values is a non-type template parameter pack
   struct value_holder {
      template <T... Values> struct apply { }; // and a pack expansion
   };

template <class... T, T... Values> // error: Values expands template type parameter
   struct static_array;            // pack T within the same template parameter list
```

—end example]

13.3 Names of template specializations

A template specialization (13.9) can be referred to by a template-id:

```cpp
template-id:
   simple-template-id:
      template-name < template-argument-list_{opt} >

   template-id:
      simple-template-id
      operator-function-id < template-argument-list_{opt} >
      literal-operator-id < template-argument-list_{opt} >

   template-name:
      identifier

   template-argument-list:
      template-argument ..._{opt}
      template-argument-list , template-argument ..._{opt}

   template-argument:
      constant-expression
      type-id
      id-expression
```
An identifier is a template-name if it is associated by name lookup with a template or an overload set that contains a function template, or the identifier is followed by <, the template-id would form an unqualified-id, and name lookup either finds one or more functions or finds nothing.

[Note 1: Whether a name actually refers to a template cannot be known in some cases until after argument dependent lookup is done (6.5.3). — end note]

When a name is considered to be a template-name, and it is followed by a <, the < is always taken as the delimiter of a template-argument-list and never as the less-than operator. When parsing a template-argument-list, the first non-nested > is taken as the ending delimiter rather than a greater-than operator. Similarly, the first non-nested >> is treated as two consecutive but distinct > tokens, the first of which is taken as the end of the template-argument-list and completes the template-id.

[Note 2: The second > token produced by this replacement rule could terminate an enclosing template-id construct or it could be part of a different construct (e.g., a cast). — end note]

[Example 1:]
```cpp
template<int i> class X { /* ... */ }
X< 1>2 > x1; // syntax error
X<(1>2)> x2; // OK

template<class T> class Y { /* ... */ }
Y<X<1>> x3; // OK, same as Y<X<1> > x3;
Y<X<6>>1>> x4; // syntax error
Y<X<(6>>1)>> x5; // OK
```

—end example

The keyword template is said to appear at the top level in a qualified-id if it appears outside of a template-argument-list or decltype-specifier. In a qualified-id of a declarator-id or in a qualified-id formed by a class-head-name (11.1) or enum-head-name (9.7.1), the keyword template shall not appear at the top level. In a qualified-id used as the name in a typename-specifier (13.8), elaborated-type-specifier (9.2.9.4), using-declaration (9.9), or class-or-decltype (11.7), an optional keyword template appearing at the top level is ignored. In these contexts, a < token is always assumed to introduce a template-argument-list. In all other contexts, when naming a template specialization of a member of an unknown specialization (13.8.3.2), the member template name shall be prefixed by the keyword template.

[Example 2:]
```cpp
struct X {
    template<std::size_t> X* alloc();
    template<std::size_t> static X* adjust();
};
template<class T> void f(T* p) {
    T* p1 = p->alloc<200>(); // error: < means less than
    T* p2 = p->template alloc<200>(); // OK: < starts template argument list
    T::adjust<100>(); // error: < means less than
    T::template adjust<100>(); // OK: < starts template argument list
}
```

—end example

A name prefixed by the keyword template shall be a template-id or the name shall refer to a class template or an alias template.

[Note 3: The keyword template cannot be applied to non-template members of class templates. — end note]

[Note 4: As is the case with the typename prefix, the template prefix is allowed in cases where it is not strictly necessary; i.e., when the nested-name-specifier or the expression on the left of the -> or . is not dependent on a template-parameter, or the use does not appear in the scope of a template. — end note]

[Example 3:]
```cpp
template <class T> struct A {
    void f(int);
    template <class U> void f(U);
};
```

132) A > that encloses the type-id of a dynamic_cast, static_cast, reinterpret_cast or const_cast, or which encloses the template-arguments of a subsequent template-id, is considered nested for the purpose of this description.
template <class T> void f(T t) {
    A<T> a;
    a.template f<t>();            // OK: calls template
    a.template f(t);             // error: not a template-id
}

template <class T> struct B {
    template <class T2> struct C { };  // OK: T::template C names a class template:
    template <class T2> struct D { };  // OK: T2::template C names a class template:
}

D<B<int> > db;  // error: no default argument for first template parameter

—end example

6 A template-id is valid if

(6.1) — there are at most as many arguments as there are parameters or a parameter is a template parameter

pack (13.7.4),

(6.2) — there is an argument for each non-deducible non-pack parameter that does not have a default template-

argument,

(6.3) — each template-argument matches the corresponding template-parameter (13.4),

(6.4) — substitution of each template argument into the following template parameters (if any) succeeds, and

(6.5) — if the template-id is non-dependent, the associated constraints are satisfied as specified in the next

paragraph.

A simple-template-id shall be valid unless it names a function template specialization (13.10.3).

[Example 4:

template<class T, T::type n = 0> class X;
struct S {
    using type = int;
};
using T1 = X<S, int, int>;  // error: too many arguments
using T2 = X<>;            // error: no default argument for first template parameter
using T3 = X<int>;         // error: value 1 does not match type-parameter
using T4 = X<int>;         // error: substitution failure for second template parameter
using T5 = X<S>;           // OK
—end example]

7 When the template-name of a simple-template-id names a constrained non-function template or a constrained

template-parameter, but not a member template that is a member of an unknown specialization

(13.8), and all template-arguments in the simple-template-id are non-dependent (13.8.3.5), the associated

constraints (13.5.3) of the constrained template shall be satisfied (13.5.2).

[Example 5:

template< typename T> concept C1 = sizeof(T) != sizeof(int);

template<C1 T> struct S1 { };  // ill-formed, no diagnostic required

template<C1 T> using PTr = T*;

S1<int> p;                   // error: constraints not satisfied
Ptr<int> p;                   // error: constraints not satisfied

template<typename T>
struct S2 { PTr<int> x; };   // OK, satisfaction is not required

template<typename T>
struct S3 { PTr<T> x; };     // error: constraints not satisfied

S3<int> x;
template<template<C1 T> class X>
struct S4 {
    X<int> x;  // ill-formed, no diagnostic required
};

template<typename T> concept C2 = sizeof(T) == 1;

template<C2 T> struct S { }

template struct S<char[2]>;  // error: constraints not satisfied

template<> struct S<char[2]>;  // error: constraints not satisfied

—end example

A concept-id is a simple-template-id where the template-name is a concept-name. A concept-id is a prvalue of type bool, and does not name a template specialization. A concept-id evaluates to true if the concept’s normalized constraint-expression (13.5.3) is satisfied (13.5.2) by the specified template arguments and false otherwise.

[Note 5: Since a constraint-expression is an unevaluated operand, a concept-id appearing in a constraint-expression is not evaluated except as necessary to determine whether the normalized constraints are satisfied. —end note]

[Example 6]:

    template<typename T> concept C = true;
    static_assert(C<int>);  // OK

—end example

13.4 Template arguments

13.4.1 General

There are three forms of template-argument, corresponding to the three forms of template-parameter: type, non-type and template. The type and form of each template-argument specified in a template-id shall match the type and form specified for the corresponding parameter declared by the template in its template-parameter-list. When the parameter declared by the template is a template parameter pack (13.7.4), it will correspond to zero or more template-arguments.

[Example 1]:

    template<class T> class Array {
        T* v;
        int sz;
    public:
        explicit Array(int);
        T& operator[](int);
        T& elem(int i) { return v[i]; }
    };

    Array<int> v1(20);
    typedef std::complex<double> dcomplex;  // std::complex is a standard library template
    Array<dcomplex> v2(30);
    Array<dcomplex> v3(40);

    void bar() {
        v1[3] = 7;
        v2[3] = v3.elem(4) = dcomplex(7,8);
    }

—end example]

In a template-argument, an ambiguity between a type-id and an expression is resolved to a type-id, regardless of the form of the corresponding template-parameter.133

[Example 2]:

    template<class T> void f();
    template<int I> void f();

133) There is no such ambiguity in a default template-argument because the form of the template-parameter determines the allowable forms of the template-argument.
The name of a template-argument shall be accessible at the point where it is used as a template-argument.

[Note 1: If the name of the template-argument is accessible at the point where it is used as a template-argument, there is no further access restriction in the resulting instantiation where the corresponding template-parameter name is used. —end note]

[Example 3]:

```cpp
template<class T> class X {
    static T t;
};

class Y {
    private:
        struct S { /* ... */ };  // X<S> is accessible
        X<S> x;  // OK: S is accessible
        // X<Y::S> has a static member of type Y::S
        // OK: even though Y::S is private
    };

    X<Y::S> y;  // error: S not accessible
};
```

For a template-argument that is a class type or a class template, the template definition has no special access rights to the members of the template-argument.

[Example 4]:

```cpp
template <template <class TT> class T> class A {
    typename T<int>::S s;
};

template <class U> class B {
    private:
        struct S { /* ... */ };  // error: A has no access to B::S
    };

    A<B> b;  // error: A has no access to B::S
};
```

When template argument packs or default template-arguments are used, a template-argument list can be empty. In that case the empty <> brackets shall still be used as the template-argument-list.

[Example 5]:

```cpp
template<class T = char> class String;
String<*> p;  // OK: String<char>
String* q;    // syntax error

template<class ... Elements> class Tuple;
Tuple<*> t;   // OK: Elements is empty
Tuple* u;     // syntax error
```

An explicit destructor call (11.4.7) for an object that has a type that is a class template specialization may explicitly specify the template-arguments.

[Example 6]:

```cpp
template<class T> struct A {
    -A();
};

void f(A<int>** p, A<int>** q) {
    p->A<int>::~A();  // OK: destructor call
};
```
6 If the use of a template-argument gives rise to an ill-formed construct in the instantiation of a template specialization, the program is ill-formed.

7 When name lookup for the name in a template-id finds an overload set, both non-template functions in the overload set and function templates in the overload set for which the template-parameters do not match the template-arguments are ignored. If none of the function templates have matching template-parameters, the program is ill-formed.

8 When a simple-template-id does not name a function, a default template-argument is implicitly instantiated (13.9.2) when the value of that default argument is needed.

[Example 7:

```cpp
template<typename T, typename U = int> struct S { }; // the type of p is S<bool, int>*
S<bool>** p;
```

The default argument for U is instantiated to form the type S<bool, int>*. —end example]

9 A template-argument followed by an ellipsis is a pack expansion (13.7.4).

### 13.4.2 Template type arguments

1 A template-argument for a template-parameter which is a type shall be a type-id.

2 [Example 1:

```cpp
template <class T> class X { };  
template <class T> void f(T t) { }
struct { } unnamed_obj;
void f() {
    struct A { };  
    enum { e1 };  
    typedef struct { } B; 
    B b;
    X<A> x1;  // OK
    X<A*> x2;  // OK
    X<B> x3;  // OK
    f(e1);  // OK
    f(unnamed_obj);  // OK
    f(b);  // OK
}
```

—end example]

[Note 1: A template type argument can be an incomplete type (6.8). —end note]

### 13.4.3 Template non-type arguments

1 If the type T of a template-parameter (13.2) contains a placeholder type (9.2.9.6) or a placeholder for a deduced class type (9.2.9.7), the type of the parameter is the type deduced for the variable x in the invented declaration

```cpp
T x = template-argument ;
```

If a deduced parameter type is not permitted for a template-parameter declaration (13.2), the program is ill-formed.

2 A template-argument for a non-type template-parameter shall be a converted constant expression (7.7) of the type of the template-parameter.

[Note 1: If the template-argument is an overload set (or the address of such, including forming a pointer-to-member), the matching function is selected from the set (12.5). —end note]

3 For a non-type template-parameter of reference or pointer type, or for each non-static data member of reference or pointer type in a non-type template-parameter of class type or subobject thereof, the reference or pointer value shall not refer to or be the address of (respectively):

(3.1) a temporary object (6.7.7),
— a string literal object (5.13.5),

— the result of a typeid expression (7.6.1.8),

— a predefined __func__ variable (9.5.1), or

— a subobject (6.7.2) of one of the above.

[Example 1:

```cpp
template<const int* pci> struct X { /* ... */ };  
X<ai> xi; // array to pointer and qualification conversions

struct Y { /* ... */ };  
template<const Y& b> struct Z { /* ... */ };  
Y y;  
Z<y> z; // no conversion, but note extra cv-qualification

template<int (&pa)[5]> struct W { /* ... */ };  
W<b> w; // no conversion

void f(char);  
void f(int);

template<void (*pf)(int)> struct A { /* ... */ };  
A<&f> a; // selects f(int)

template<auto n> struct B { /* ... */ };  
B<5> b1; // OK, template parameter type is int  
B<'a'> b2; // OK, template parameter type is char  
B<2.5> b3; // OK, template parameter type is double  
B<void(0)> b4; // error: template parameter type cannot be void
```
— end example]

[Note 2: A string-literal (5.13.5) is not an acceptable template-argument for a template-parameter of non-class type.

[Example 2:

```cpp
template<class T, T p> class X { /* ... */ }

X<const char*, "Studebaker"> x; // error: string literal object as template-argument  
X<const char*, "Knope" + 1> x2; // error: subobject of string literal object as template-argument

const char p[] = "Vivisectionist";  
X<const char*, p> y; // OK

struct A {
    constexpr A(const char*) {}  
};

X<A, "Pyrophoricity"> z; // OK, string-literal is a constructor argument to A
— end example]

— end note]

[Note 3: A temporary object is not an acceptable template-argument when the corresponding template-parameter has reference type.

[Example 3:

```cpp
template<const int& CRI> struct B { /* ... */ };  
B<1> b1; // error: temporary would be required for template argument

int c = 1;
```
]
```
B<< b2;     // OK
struct X { int n; };
struct Y { const int &r; };  
template<Y y> struct C { /* ... */ };   
C<Y<X(1).n>> c;       // error: subobject of temporary object used to initialize
                      // reference member of template parameter
```

--- end example

--- end note

13.4.4 Template template arguments

1 A template-argument for a template template-parameter shall be the name of a class template or an alias template, expressed as id-expression. When the template-argument names a class template, only primary class templates are considered when matching the template template argument with the corresponding parameter; partial specializations are not considered even if their parameter lists match that of the template template parameter.

2 Any partial specializations (13.7.6) associated with the primary class template or primary variable template are considered when a specialization based on the template template-parameter is instantiated. If a specialization is not visible at the point of instantiation, and it would have been selected had it been visible, the program is ill-formed, no diagnostic required.

Example 1:
```
template<class T> class A { // primary template
  int x;
};
template<class T> class A<T*> { // partial specialization
  long x;
};
template<template<class U> class V> class C { 
  V<int> y;
  V<int*> z;
};
C<A> c;         // V<int> within C<A> uses the primary template, so c.y.x has type int
                // V<int*> within C<A> uses the partial specialization, so c.z.x has type long
```

--- end example

3 A template-argument matches a template template-parameter P when P is at least as specialized as the template-argument A. In this comparison, if P is unconstrained, the constraints on A are not considered. If P contains a template parameter pack, then A also matches P if each of A’s template parameters matches the corresponding template parameter in the template-head of P. Two template parameters match if they are of the same kind (type, non-type, template), for non-type template-parameters, their types are equivalent (13.7.7.2), and for template template-parameters, each of their corresponding template parameters matches recursively. When P’s template-head contains a template parameter pack (13.7.4), the template parameter pack will match zero or more template parameters or template parameter packs in the template-head of A with the same type and form as the template parameter pack in P (ignoring whether those template parameters are template parameter packs).

Example 2:
```
template<class T> class A { /* ... */ };  
template<class T, class U = T> class B { /* ... */ };   
template<class ... Types> class C { /* ... */ };   
template<auto n> class D { /* ... */ };     
template<template<class> class P> class X { /* ... */ };   
template<template<class ...> class P> class Y { /* ... */ };   
template<template<int> class R> class Z { /* ... */ };   
```

X<A> xa;      // OK
X<B> xb;      // OK
X<C> xc;      // OK
Y<A> ya;      // OK
Y<B> yb;      // OK
Example 3:

```
template <class T> struct eval;

template <template <class, class...> class TT, class T1, class... Rest>
struct eval<TT<T1, Rest...>> { };

template <class T1> struct A;
template <class T1, class T2> struct B;
template <int N> struct C;
template <class T1, int N> struct D;
template <class T1, class T2, int N = 17> struct E;

eval<A<int>> eA; // OK: matches partial specialization of eval

eval<B<int, float>> eB; // OK: matches partial specialization of eval

eval<C<int, T1>> eC; // error: C does not match TT in partial specialization

eval<D<int, float>> eD; // error: D does not match TT in partial specialization

eval<E<int, float>> eE; // error: E does not match TT in partial specialization
```

Example 4:

```
template<concept C> concept C = requires (T t) { t.f(); };
template<concept C> concept D = C<T> && requires (T t) { t.g(); };

template<concept C> struct S { };
template<concept C> struct X { };
template<concept C> struct Y { };
template<concept C> struct Z { };

S<X> s1; // OK, X and P have equivalent constraints
S<Y> s2; // error: P is not at least as specialized as Y
S<Z> s3; // OK, P is at least as specialized as Z
```

--- end example

A template template-parameter P is at least as specialized as a template template-argument A if, given the following rewrite to two function templates, the function template corresponding to P is at least as specialized as the function template corresponding to A according to the partial ordering rules for function templates (13.7.7.3). Given an invented class template X with the template-head of A (including default arguments and requires-clause, if any):

1. Each of the two function templates has the same template parameters and requires-clause (if any), respectively, as P or A.
2. Each function template has a single function parameter whose type is a specialization of X with template arguments corresponding to the template parameters from the respective function template where, for each template parameter PP in the template-head of the function template, a corresponding template argument AA is formed. If PP declares a template parameter pack, then AA is the id-expression PP... (13.7.4); otherwise, AA is the id-expression PP.

If the rewrite produces an invalid type, then P is not at least as specialized as A.

### 13.5 Template constraints

#### 13.5.1 General

[Note 1: Subclause 13.5 defines the meaning of constraints on template arguments. The abstract syntax and satisfaction rules are defined in 13.5.2. Constraints are associated with declarations in 13.5.3. Declarations are partially ordered by their associated constraints (13.5.5).]
### 13.5.2 Constraints

#### 13.5.2.1 General

A **constraint** is a sequence of logical operations and operands that specifies requirements on template arguments. The operands of a logical operation are constraints. There are three different kinds of constraints:

1. **Conjunctions** (13.5.2.2),
2. **Disjunctions** (13.5.2.2), and
3. **Atomic constraints** (13.5.2.3).

In order for a constrained template to be instantiated (13.9), its associated constraints (13.5.3) shall be satisfied as described in the following subclauses.

[Note 1: Forming the name of a specialization of a class template, a variable template, or an alias template (13.3) requires the satisfaction of its constraints. Overload resolution (12.4.3) requires the satisfaction of constraints on functions and function templates. — end note]

#### 13.5.2.2 Logical operations

There are two binary logical operations on constraints: conjunction and disjunction.

[Note 1: These logical operations have no corresponding C++ syntax. For the purpose of exposition, conjunction is spelled using the symbol \( \wedge \) and disjunction is spelled using the symbol \( \vee \). The operands of these operations are called the left and right operands. In the constraint \( A \wedge B \), \( A \) is the left operand, and \( B \) is the right operand. — end note]

A **conjunction** is a constraint taking two operands. To determine if a conjunction is satisfied, the satisfaction of the first operand is checked. If that is not satisfied, the conjunction is not satisfied. Otherwise, the conjunction is satisfied if and only if the second operand is satisfied.

A **disjunction** is a constraint taking two operands. To determine if a disjunction is satisfied, the satisfaction of the first operand is checked. If that is satisfied, the disjunction is satisfied. Otherwise, the disjunction is satisfied if and only if the second operand is satisfied.

[Example 1:]

```cpp
template<typename T>
const expr bool get_value() { return T::value; }

template<typename T>
requires (sizeof(T) > 1) && (get_value<T>())
void f(T); // has associated constraint sizeof(T) > 1 \&\& get_value<T>()
void f(int);

f('a'); // OK: calls f(int)
```

In the satisfaction of the associated constraints (13.5.3) of \( f \), the constraint `sizeof(char) > 1` is not satisfied; the second operand is not checked for satisfaction. — *end example*

[Note 2: A logical negation expression (7.6.2.2) is an atomic constraint; the negation operator is not treated as a logical operation on constraints. As a result, distinct negation constraint-expressions that are equivalent under 13.7.7.2 do not subsume one another under 13.5.5. Furthermore, if substitution to determine whether an atomic constraint is satisfied (13.5.2.3) encounters a substitution failure, the constraint is not satisfied, regardless of the presence of a negation operator.

[Example 2:]

```cpp
template <class T> concept sad = false;

template <class T> int f1(T) requires (!sad<T>);
template <class T> int f1(T) requires (!sad<T>) && true;
int i1 = f1(42); // ambiguous, !sad<T> atomic constraint expressions (13.5.2.3)
 // are not formed from the same expression

template <class T> concept not_sad = !sad<T>;
template <class T> int f2(T) requires not_sad<T>;
template <class T> int f2(T) requires not_sad<T> && true;
int i2 = f2(42); // OK, !sad<T> atomic constraint expressions both come from not_sad
```

§ 13.5.2.2
template <class T> int f3(T) requires (!sad<typename T::type>);
int i3 = f3(42); // error: associated constraints not satisfied due to substitution failure

template <class T> concept sad_nested_type = sad<typename T::type>;

int i4 = f4(42); // OK, substitution failure contained within sad_nested_type

Here, requires (!sad<typename T::type>) requires that there is a nested type that is not sad, whereas requires (!sad_nested_type<T>) requires that there is no sad nested type. —end example

13.5.2.3 Atomic constraints

An atomic constraint is formed from an expression E and a mapping from the template parameters that appear within E to template arguments that are formed via substitution during constraint normalization in the declaration of a constrained entity (and, therefore, can involve the unsubstituted template parameters of the constrained entity), called the parameter mapping (13.5.3).

[Note 1: Atomic constraints are formed by constraint normalization (13.5.4). E is never a logical AND expression (7.6.14) nor a logical OR expression (7.6.15).—end note]

Two atomic constraints, e1 and e2, are identical if they are formed from the same appearance of the same expression and if, given a hypothetical template A whose template-parameter-list consists of template-parameters corresponding and equivalent (13.7.7.2) to those mapped by the parameter mappings of the expression, a template-id naming A whose template-arguments are the targets of the parameter mapping of e1 is the same (13.6) as a template-id naming A whose template-arguments are the targets of the parameter mapping of e2.

[Note 2: The comparison of parameter mappings of atomic constraints operates in a manner similar to that of declaration matching with alias template substitution (13.7.8).]

[Example 1:]

```cpp
template <unsigned N> constexpr bool Atomic = true;
template <unsigned N> concept C = Atomic<N>;
template <unsigned N> concept Add1 = C<N+1>;
template <unsigned N> concept AddOne = C<N+1>;
template <unsigned N> void f()
  requires Add1<2 * M>;
template <unsigned N> int f()
  requires AddOne<2 * M> && true;

int x = f<0>(); // OK, the atomic constraints from concept C in both f's are Atomic<N>
  // with mapping similar to N -> 2 * M + 1
```

```
template <unsigned N> struct WrapN;
template <unsigned N> using Add1Ty = WrapN<N+1>;
template <unsigned N> using AddOneTy = WrapN<N+1>;
template <unsigned N> void g(Add1Ty<2 * M> *);
template <unsigned N> void g(AddOneTy<2 * M> *);
```

```cpp
void h() {
g<0>(nullptr); // OK, there is only one g
}
```

—end example]

This similarity includes the situation where a program is ill-formed, no diagnostic required, when the meaning of the program depends on whether two constructs are equivalent, and they are functionally equivalent but not equivalent.

[Example 2:]

```cpp
template <unsigned N> void f2()
  requires Add1<2 * M>;
template <unsigned N> int f2()
  requires Add1<2 * M> && true;
void h2() {
f2<0>(); // ill-formed, no diagnostic required:
```
To determine if an atomic constraint is satisfied, the parameter mapping and template arguments are first substituted into its expression. If substitution results in an invalid type or expression, the constraint is not satisfied. Otherwise, the lvalue-to-rvalue conversion (7.3.2) is performed if necessary, and \( E \) shall be a constant expression of type \( \text{bool} \). The constraint is satisfied if and only if evaluation of \( E \) results in \( \text{true} \).

If, at different points in the program, the satisfaction result is different for identical atomic constraints and template arguments, the program is ill-formed, no diagnostic required.

**Example 3:**

```cpp
template<typename T> concept C =
    sizeof(T) == 4 && !true;  // requires atomic constraints sizeof(T) == 4 and !true

template<typename T> struct S {
    constexpr operator bool() const { return true; }
};

template<typename T> requires (S<T>{})
    void f(T);  // #1
    void f(int);  // #2

void g() {
    f(0);  // error: expression S<int>{} does not have type bool
    // while checking satisfaction of deduced arguments of #1;
    // call is ill-formed even though #2 is a better match
}
```

### 13.5.3 Constrained declarations

A template declaration (13.1) or templated function declaration (9.3.4.6) can be constrained by the use of a `requires-clause`. This allows the specification of constraints for that declaration as an expression:

```
constraint-expression:
    logical-or-expression
```

Constraints can also be associated with a declaration through the use of `type-constraints` in a `template-parameter-list` or parameter-type-list. Each of these forms introduces additional `constraint-expressions` that are used to constrain the declaration.

A declaration’s associated constraints are defined as follows:

1. If there are no introduced `constraint-expressions`, the declaration has no associated constraints.
2. Otherwise, if there is a single introduced `constraint-expression`, the associated constraints are the normal form (13.5.4) of that expression.
3. Otherwise, the associated constraints are the normal form of a logical AND expression (7.6.14) whose operands are in the following order:

   1. the `constraint-expression` introduced by each `type-constraint` (13.2) in the declaration’s `template-parameter-list`, in order of appearance, and
   2. the `constraint-expression` introduced by a `requires-clause` following a `template-parameter-list` (13.1), and
   3. the `constraint-expression` introduced by each `type-constraint` in the parameter-type-list of a function declaration, and
   4. the `constraint-expression` introduced by a trailing `requires-clause` (9.3) of a function declaration (9.3.4.6).

The formation of the associated constraints establishes the order in which constraints are instantiated when checking for satisfaction (13.5.2).

**Example 1:**
template<typename T> concept C = true;

template<C T> void f1(T);
template<typename T> requires C<T> void f2(T);
template<typename T> void f3(T) requires C<T>;

The functions f1, f2, and f3 have the associated constraint C<T>.

template<typename T> concept C1 = true;
template<typename T> concept C2 = sizeof(T) > 0;
template<C1 T> requires C2<T> void f4(T);
template<typename T> requires C1<T> && C2<T> void f5(T);

The associated constraints of f4 and f5 are C1<T> ∧ C2<T>.

template<C1 T> requires C2<T> void f6();
template<C2 T> requires C1<T> void f7();

The associated constraints of f6 are C1<T> ∧ C2<T>, and those of f7 are C2<T> ∧ C1<T>. — end example]

4 When determining whether a given introduced constraint-expression C1 of a declaration in an instantiated specialization of a templated class is equivalent (13.7.7.2) to the corresponding constraint-expression C2 of a declaration outside the class body, C1 is instantiated. If the instantiation results in an invalid expression, the constraint-expressions are not equivalent.

[Note 1: This can happen when determining which member template is specialized by an explicit specialization declaration. — end note]

[Example 2:

template <class T> concept C = true;
template <class T> struct A {
    template <class U> U f(U) requires C<typename T::type>; // #1
    template <class U> U f(U) requires C<T>; // #2
};

template <> template <class U>
U A<int>::f(U u) requires C<int> { return u; } // OK, specializes #2

Substituting int for T in C<typename T::type> produces an invalid expression, so the specialization does not match #1. Substituting int for T in C<T> produces C<int>, which is equivalent to the constraint-expression for the specialization, so it does match #2. — end example]

13.5.4 Constraint normalization [temp.constr.normal]

The normal form of an expression E is a constraint (13.5.2) that is defined as follows:

(1.1) — The normal form of an expression ( E ) is the normal form of E.

(1.2) — The normal form of an expression E1 || E2 is the disjunction (13.5.2.2) of the normal forms of E1 and E2.

(1.3) — The normal form of an expression E1 && E2 is the conjunction of the normal forms of E1 and E2.

(1.4) — The normal form of a concept-id C<A1, A2, ..., An> is the normal form of the constraint-expression of C, after substituting A1, A2, ..., An for C’s respective template parameters in the parameter mappings in each atomic constraint. If any such substitution results in an invalid type or expression, the program is ill-formed; no diagnostic is required.

[Example 1:

template<typename T> concept A = T::value || true;
template<typename U> concept B = A<U*>;
template<typename V> concept C = B<V&*>;

Normalization of B’s constraint-expression is valid and results in T::value (with the mapping T → U*) ∨ true (with an empty mapping), despite the expression T::value being ill-formed for a pointer type T. Normalization of C’s constraint-expression results in the program being ill-formed, because it would form the invalid type V&* in the parameter mapping. — end example]

(1.5) — The normal form of any other expression E is the atomic constraint whose expression is E and whose parameter mapping is the identity mapping.

§ 13.5.4 378
The process of obtaining the normal form of a constraint-expression is called normalization.

[Note 1: Normalization of constraint-expressions is performed when determining the associated constraints (13.5.2) of a declaration and when evaluating the value of an id-expression that names a concept specialization (7.5.4). — end note]

[Example 2:

```cpp
template<typename T> concept C1 = sizeof(T) == 1;
template<typename T> concept C2 = C1<T> && 1 == 2;
template<typename T> concept C3 = requires { typename T::type; }; // #1
template<typename T> concept C4 = requires { requires(T x) { ++x; } }; // #2

template<C2 U> void f1(U); // #1
template<C3 U> void f2(U); // #2
template<C4 U> void f3(U); // #3
```

The associated constraints of #1 are `sizeof(T) == 1` (with mapping `T` to `U`).
The associated constraints of #2 are `requires { typename T::type; }` (with mapping `T` to `U`).
The associated constraints of #3 are `requires (T x) { ++x; }` (with mapping `T` to `U`). — end example]

13.5.5 Partial ordering by constraints

A constraint \( P \) subsumes a constraint \( Q \) if and only if, for every disjunctive clause \( P_i \) in the disjunctive normal form\(^\text{135}\) of \( P \), \( P_i \) subsumes every conjunctive clause \( Q_j \) in the conjunctive normal form\(^\text{134}\) of \( Q \), where

\[ \forall i,j \text{ s.t. } P_i \text{ is a disjunctive clause of } P \text{ and } Q_j \text{ is a conjunctive clause of } Q : \]

\[ \text{if and only if there exists an atomic constraint } P_{ia} \text{ in } P_i \text{ for which there exists an atomic constraint } Q_{jb} \text{ in } Q_j \text{ such that } P_{ia} \text{ subsumes } Q_{jb} \text{, and} \]

\[ \text{if and only if there exists an atomic constraint } A \text{ in } P \text{ for which there exists an atomic constraint } B \text{ in } Q \text{ such that } A \text{ subsumes } B \text{. Also note that every constraint subsumes itself. — end example}

[Example 1: Let \( A \) and \( B \) be atomic constraints (13.5.2.3). The constraint \( A \land B \) subsumes \( A \), but \( A \) does not subsume \( A \lor B \). The constraint \( A \) subsumes \( A \lor B \), but \( A \lor B \) does not subsume \( A \). Also note that every constraint subsumes itself. — end example]

1. A constraint \( P \) subsumes a constraint \( Q \) if and only if, for every disjunctive clause \( P_i \) in the disjunctive normal form\(^\text{134}\) of \( P \), \( P_i \) subsumes every conjunctive clause \( Q_j \) in the conjunctive normal form\(^\text{135}\) of \( Q \), where

\[ \forall i,j \text{ s.t. } P_i \text{ is a disjunctive clause of } P \text{ and } Q_j \text{ is a conjunctive clause of } Q : \]

\[ \exists i.a, j.b \text{ s.t. } \exists P_{i.a} \text{ in } P_i \text{ for which } \exists Q_{j.b} \text{ in } Q_j \text{ such that } P_{i.a} \text{ subsumes } Q_{j.b} \text{, and} \]

\[ \exists a \text{ s.t. } \exists A \text{ in } P \text{ for which } \exists B \text{ in } Q \text{ such that } A \text{ subsumes } B \text{. Also note that every constraint subsumes itself. — end example]

2. A declaration \( D_1 \) is at least as constrained as a declaration \( D_2 \) if

\[ \forall i \text{ s.t. } D_1 \text{ and } D_2 \text{ are both constrained declarations and } D_1 \text{‘s associated constraints subsume those of } D_2 \text{; or} \]

\[ \exists P \text{ s.t. } D_2 \text{ has no associated constraints.} \]

3. A declaration \( D_1 \) is more constrained than another declaration \( D_2 \) when \( D_1 \) is at least as constrained as \( D_2 \), and \( D_2 \) is not at least as constrained as \( D_1 \).

[Example 2:

```cpp
template<typename T> concept C1 = requires(T t) { -t; }; // #1
template<typename T> concept C2 = C1<T> && requires(T t) { *t; }; // #2

template<C1 T> void f(T); // #1
template<C2 T> void f(T); // #2
template<typename T> void g(T); // #3
template<C1 T> void g(T); // #4
```

— end note

134) A constraint is in disjunctive normal form when it is a disjunction of clauses where each clause is a conjunction of atomic constraints. For atomic constraints \( A \), \( B \), and \( C \), the disjunctive normal form of the constraint \( A \land (B \lor C) \) is \( (A \land B) \lor (A \land C) \). Its disjunctive clauses are \( (A \land B) \) and \( (A \land C) \).

135) A constraint is in conjunctive normal form when it is a conjunction of clauses where each clause is a disjunction of atomic constraints. For atomic constraints \( A \), \( B \), and \( C \), the constraint \( A \land (B \lor C) \) is in conjunctive normal form. Its conjunctive clauses are \( A \lor (B \lor C) \).
f(0); // selects #1
f((int*)0); // selects #2
g(true); // selects #3 because C1<bool> is not satisfied
g(0); // selects #4
— end example]

13.6 Type equivalence

Two template-ids are the same if
1.1 — their template-names, operator-function-ids, or literal-operator-ids refer to the same template, and
1.2 — their corresponding type template-arguments are the same type, and
1.3 — their corresponding non-type template-arguments are template-argument-equivalent (see below) after conversion to the type of the template-parameter, and
1.4 — their corresponding template template-arguments refer to the same template.

Two template-ids that are the same refer to the same class, function, or variable.

Two values are template-argument-equivalent if they are of the same type and
2.1 — they are of integral type and their values are the same, or
2.2 — they are of floating-point type and their values are identical, or
2.3 — they are of type std::nullptr_t, or
2.4 — they are of enumeration type and their values are the same, or
2.5 — they are of pointer type and they have the same pointer value, or
2.6 — they are of pointer-to-member type and they refer to the same class member or are both the null member pointer value, or
2.7 — they are of reference type and they refer to the same object or function, or
2.8 — they are of array type and their corresponding elements are template-argument-equivalent, or
2.9 — they are of union type and either they both have no active member or they have the same active member and their active members are template-argument-equivalent, or
2.10 — they are of class type and their corresponding direct subobjects and reference members are template-argument-equivalent.

[Example 1:

    template<class E, int size> class buffer { /* ... */ };  
    buffer<char,2*512> x;  
    buffer<char,1024> y;  

declares x and y to be of the same type, and

    template<class T, void(*err_fct)()> class list { /* ... */ };  
    list<int,&error_handler1> x1;  
    list<int,&error_handler2> x2;  
    list<int,&error_handler2> x3;  
    list<char,&error_handler2> x4;  

declares x2 and x3 to be of the same type. Their type differs from the types of x1 and x4.

    template<class T> struct X { };  
    template<class> struct Y { };  
    template<class T> struct Y<Z<int>> x;  
    x.z=y;  

declares y and z to be of the same type. — end example]

4 If an expression e is type-dependent (13.8.3.3), decltype(e) denotes a unique dependent type. Two such decltype-specifiers refer to the same type only if their expressions are equivalent (13.7.7.2).

[Note 1: However, such a type might be aliased, e.g., by a typedef-name. — end note]

—end example]

136) The identity of enumerators is not preserved.
137) An array as a template-parameter decays to a pointer.
13.7 Template declarations

13.7.1 General

A template-id, that is, the template-name followed by a template-argument-list shall not be specified in the declaration of a primary template declaration.

[Example 1:

```cpp
template<class T1, class T2, int I> class A<T1, T2, I> { }; // error
template<class T1, int I> void sort<T1, I>(T1 data[I]); // error
```
—end example]

[Note 1: However, this syntax is allowed in class template partial specializations (13.7.6). —end note]

For purposes of name lookup and instantiation, default arguments, type-constraints, requires-clauses (13.1), and noexcept-specifiers of function templates and of member functions of class templates are considered definitions; each default argument, type-constraint, requires-clause, or noexcept-specifier is a separate definition which is unrelated to the templated function definition or to any other default arguments type-constraints, requires-clauses, or noexcept-specifiers. For the purpose of instantiation, the substatements of a constexpr if statement (8.5.2) are considered definitions.

Because an alias-declaration cannot declare a template-id, it is not possible to partially or explicitly specialize an alias template.

13.7.2 Class templates

13.7.2.1 General

A class template defines the layout and operations for an unbounded set of related types.

[Example 1: A single class template List might provide an unbounded set of class definitions: one class List<T> for every type T, each describing a linked list of elements of type T. Similarly, a class template Array describing a contiguous, dynamic array might be defined like this:

```cpp
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
};
```

The prefix template<class T> specifies that a template is being declared and that a type-name T may be used in the declaration. In other words, Array is a parameterized type with T as its parameter. —end example]

When a member function, a member class, a member enumeration, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the template-head is equivalent to that of the class template (13.7.7.2). The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list.

[Example 2:

```cpp
template<class T1, class T2> struct A {
    void f1();
    void f2();
};

template<class T2, class T1> void A<T2,T1>::f1() { } // OK
template<class T2, class T1> void A<T1,T2>::f2() { } // error

template<class ... Types> struct B {
    void f3();
    void f4();
};
```
In a redeclaration, partial specialization, explicit specialization or explicit instantiation of a class template, the class-key shall agree in kind with the original class template declaration (9.2.9.4).

13.7.2.2 Member functions of class templates

A member function of a class template may be defined outside of the class template definition in which it is declared.

[Example 1:

```cpp
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
};
```

declares three member functions of a class template. The subscript function might be defined like this:

```cpp
template<class T> T& Array<T>::operator[](int i) {
    if (i<0 || sz<=i) error("Array: range error");
    return v[i];
}
```

A constrained member function can be defined out of line:

```cpp
template<typename T> concept C = requires {
    typename T::type;
};

template<typename T> struct S {
    void f() requires C<T>;
    void g() requires C<T>;
};

template<typename T>
void S<T>::f() requires C<T> { }  // OK

template<typename T>
void S<T>::g() { }  // error: no matching function in S<T>
```

—end example]
Example 2: The template-argument for Array<T>::operator[] will be determined by the Array to which the subscripting operation is applied.

```cpp
Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7; // Array<int>::operator[]
v2[3] = dcomplex(7,8); // Array<dcomplex>::operator[]
```

---end example---

13.7.2.3 Deduction guides

Deduction guides are used when a template-name appears as a type specifier for a deduced class type (9.2.9.7). Deduction guides are not found by name lookup. Instead, when performing class template argument deduction (12.4.2.9), any deduction guides declared for the class template are considered.

```cpp
deduction-guide:
  explicit-specifier_opt template-name ( parameter-declaration-clause ) -> simple-template-id ;
```

Example 1:

```cpp
template<class T, class D = int>
struct S {
  T data;
};
template<class U>
S(U) -> S<typename U::type>;

struct A {
  using type = short;
  operator type();
};
S x(A()); // x is of type S<short, int>
```

---end example---

13.7.2.4 Member classes of class templates

A member class of a class template may be defined outside the class template definition in which it is declared. [Note 1: The member class must be defined before its first use that requires an instantiation (13.9.2). For example,]

```cpp
template<class T> struct A {
  class B;
};
A<int>::B* b1; // OK: requires A to be defined but not A::B
template<class T> class A<T>::B { }; // OK: requires A::B to be defined
```

---end note---

13.7.2.5 Static data members of class templates

A definition for a static data member or static data member template may be provided in a namespace scope enclosing the definition of the static member's class template.

Example 1:

```cpp
template<class T> class X {
  static T s;
};
template<class T> T X<T>::s = 0;
```
struct limits {
    template<class T>
    static const T min; // declaration
};

    template<class T>
    const T limits::min = { }; // definition
— end example]

An explicit specialization of a static data member declared as an array of unknown bound can have a different
bound from its definition, if any.

[Example 2:
    template <class T> struct A {
        static int i[];
    };  
    template <class T> int A<T>::i[4];  // 4 elements
    template <> int A<int>::i[] = { 1 };  // OK: 1 element
— end example]

13.7.2.6 Enumeration members of class templates  

An enumeration member of a class template may be defined outside the class template definition.

[Example 1:
    template<class T> struct A {
        enum E : T;
    };  
    A<int> a;
    template<class T> enum A<T>::E : T { e1, e2 };  
    A<int>::E e = A<int>::e1;
— end example]

13.7.3 Member templates  

A template can be declared within a class or class template; such a template is called a member template. A
member template can be defined within or outside its class definition or class template definition. A member
template of a class template that is defined outside of its class template definition shall be specified with a
template-head equivalent to that of the class template followed by a template-head equivalent to that of the
member template (13.7.7.2).

[Example 1:
    template<class T> struct string {
        template<class T2> int compare(const T2&);
        template<class T2> string(const string<T2>& s) { /* ... */ } 
    };
    template<class T> template<class T2> int string<T>::compare(const T2& s) {
    }  
— end example]

[Example 2:
    template<typename T> concept C1 = true;
    template<typename T> concept C2 = sizeof(T) <= 4;
    template<C1 T> struct S {
        template<C2 U> void f(U);
        template<C2 U> void g(U);
    };
    template<C1 T> template<C2 U>
    void S<T>::f(U) { }  // OK
    template<C1 T> template<typename U>
    void S<T>::g(U) { }  // error: no matching function in S<T> 

§ 13.7.3 384
A local class of non-closure type shall not have member templates. Access control rules (11.9) apply to member template names. A destructor shall not be a member template. A non-template member function (9.3.4.6) with a given name and type and a member function template of the same name, which could be used to generate a specialization of the same type, can both be declared in a class. When both exist, a use of that name and type refers to the non-template member unless an explicit template argument list is supplied.

Example 3:

```c++
template <class T> struct A {
  void f(int);
  template <class T2> void f(T2);
};

template <> void A<int>::f(int) { } // non-template member function
template <> template <> void A<int>::f<int>(int) { } // member function template specialization

int main() {
  A<char> ac;
  ac.f(1); // non-template
  ac.f('c'); // template
  ac.f<int>(1); // template
}
```

Example 4:

```c++
template <class T> struct AA {
  template <class C> virtual void g(C);
  virtual void f(); // OK
};
```

Example 5:

```c++
class B {
  virtual void f(int);
};

class D : public B {
  template <class T> void f(T); // does not override B::f(int)
  void f(int i) { f<int>(i); } // overriding function that calls the function template specialization
};
```

Example 6:

```c++
struct A {
  template <class T> operator T*();
};

template <class T> A::operator T*(){ return 0; }
template <> A::operator char*(){ return 0; } // specialization
template A::operator void*(); // explicit instantiation

int main() {
  A a;
  int* ip;
  ip = a.operator int*(); // explicit call to template operator A::operator int*()
}
```
Note 1: There is no syntax to form a template-id (13.3) by providing an explicit template argument list (11.10.2) for a conversion function template (11.4.8.3). — end note

A specialization of a conversion function template is not found by name lookup. Instead, any conversion function templates visible in the context of the use are considered. For each such operator, if argument deduction succeeds (13.10.3.4), the resulting specialization is used as if found by name lookup.

A using-declaration in a derived class cannot refer to a specialization of a conversion function template in a base class.

Overload resolution (12.4.4.3) and partial ordering (13.7.7.3) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

13.7.4 Variadic templates

A template parameter pack is a template parameter that accepts zero or more template arguments.

Example 1:

```cpp
template<class ... Types> struct Tuple { }

Tuple<> t0; // Types contains no arguments
Tuple<int> t1; // Types contains one argument: int
Tuple<int, float> t2; // Types contains two arguments: int and float
Tuple<> error; // error: 0 is not a type
```

- end example

A function parameter pack is a function parameter that accepts zero or more function arguments.

Example 2:

```cpp
template<class ... Types> void f(Types ... args);

f(); // args contains no arguments
f(1); // args contains one argument: int
f(2, 1.0); // args contains two arguments: int and double
```

- end example

An init-capture pack is a lambda capture that introduces an init-capture for each of the elements in the pack expansion of its initializer.

Example 3:

```cpp
template <typename... Args>
void foo(Args... args) {
    [...xs=args] {
        bar(xs...); // xs is an init-capture pack
    }
}
```

```cpp
foo(); // xs contains zero init-captures
foo(1); // xs contains one init-capture
```

- end example

A pack is a template parameter pack, a function parameter pack, or an init-capture pack. The number of elements of a template parameter pack or a function parameter pack is the number of arguments provided for the parameter pack. The number of elements of an init-capture pack is the number of elements in the pack expansion of its initializer.

A pack expansion consists of a pattern and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

1. In a function parameter pack (9.3.4.6); the pattern is the parameter-declaration without the ellipsis.
2. In a using-declaration (9.9); the pattern is a using-declarator.
3. In a template parameter pack that is a pack expansion (13.2):
   1. if the template parameter pack is a parameter-declaration; the pattern is the parameter-declaration without the ellipsis;
— if the template parameter pack is a type-parameter; the pattern is the corresponding type-parameter without the ellipsis.

— In an initializer-list (9.4); the pattern is an initializer-clause.

— In a base-specifier-list (11.7); the pattern is a base-specifier.

— In a mem-initializer-list (11.10.3) for a mem-initializer whose mem-initializer-id denotes a base class; the pattern is the mem-initializer.

— In a template-argument-list (13.4); the pattern is a template-argument.

— In an attribute-list (9.12.1); the pattern is an attribute.

— In an alignment-specifier (9.12.2); the pattern is the alignment-specifier without the ellipsis.

— In a capture-list (7.5.5.3); the pattern is the capture without the ellipsis.

— In a sizeof... expression (7.6.2.5); the pattern is an identifier.

— In a fold-expression (7.5.6); the pattern is the cast-expression that contains an unexpanded pack.

[Example 4:

```cpp
template<class ... Types> void f(Types ... rest);
template<class ... Types> void g(Types ... rest) {
  f(&rest ...);  // “&rest ...” is a pack expansion; “&rest” is its pattern
}
```

— end example]

6 For the purpose of determining whether a pack satisfies a rule regarding entities other than packs, the pack is considered to be the entity that would result from an instantiation of the pattern in which it appears.

7 A pack whose name appears within the pattern of a pack expansion is expanded by that pack expansion. An appearance of the name of a pack is only expanded by the innermost enclosing pack expansion. The pattern of a pack expansion shall name one or more packs that are not expanded by a nested pack expansion; such packs are called unexpanded packs in the pattern. All of the packs expanded by a pack expansion shall have the same number of arguments specified. An appearance of a name of a pack that is not expanded is ill-formed.

[Example 5:

```cpp
template<typename...> struct Tuple {};  
template<typename T1, typename T2> struct Pair {};  

template<class ... Args1> struct zip {  
template<class ... Args2> struct with {
  typedef Tuple<Pair<Args1, Args2> ... > type;
};
};

typedef zip<short, int>::with<unsigned short, unsigned>::type T1;  // T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned>>
typedef zip<short>::with<unsigned short, unsigned>::type T2;  // error: different number of arguments specified for Args1 and Args2

void g(Args ... args) {
  // OK: Args is expanded by the function parameter pack args
  f(const_cast<const Args*>(&args)...);  // OK: “Args” and “args” are expanded
  f(5 ...);  // error: pattern does not contain any packs
  f(args);  // error: pack “args” is not expanded
  f(b(args ...) + args ...);  // OK: first “args” expanded within b,
                             // second “args” expanded within f
}
```

— end example]

8 The instantiation of a pack expansion that is neither a sizeof... expression nor a fold-expression produces a list of elements $E_1, E_2, \ldots, E_N$, where $N$ is the number of elements in the pack expansion parameters. Each $E_i$ is generated by instantiating the pattern and replacing each pack expansion parameter with its $i$th element. Such an element, in the context of the instantiation, is interpreted as follows:
if the pack is a template parameter pack, the element is a template parameter (13.2) of the corresponding kind (type or non-type) designating the \(i\)th corresponding type or value template argument;

— if the pack is a function parameter pack, the element is an id-expression designating the \(i\)th function parameter that resulted from instantiation of the function parameter pack declaration; otherwise

— if the pack is an init-capture pack, the element is an id-expression designating the variable introduced by the \(i\)th init-capture that resulted from instantiation of the init-capture pack.

All of the \(E_i\) become items in the enclosing list.

[Note 1: The variety of list varies with the context: expression-list, base-specifier-list, template-argument-list, etc.  
end note]

When \(N\) is zero, the instantiation of the expansion produces an empty list. Such an instantiation does not alter the syntactic interpretation of the enclosing construct, even in cases where omitting the list entirely would otherwise be ill-formed or would result in an ambiguity in the grammar.

[Example 6:]

```cpp
template<class... T> struct X : T... { };  
template<class... T> void f(T... values) {
    X<T...> x(values...);
}

template void f<>();  // OK: X<\> has no base classes
                      // x is a variable of type X<\> that is value-initialized
```

[Example 7:]

```cpp
template<typename ...Args>
bool all(Args ...args) { return (... && args); }

bool b = all(true, true, true, false);
```

Within the instantiation of all, the returned expression expands to \((true && true) && true) && false\), which evaluates to false.  — end example]

If \(N\) is zero for a unary fold-expression, the value of the expression is shown in Table 17; if the operator is not listed in Table 17, the instantiation is ill-formed.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Value when pack is empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>void()</td>
</tr>
</tbody>
</table>

Table 17: Value of folding empty sequences  [tab:temp.fold.empty]
13.7.5 Friends

A friend of a class or class template can be a function template or class template, a specialization of a function template or class template, or a non-template function or class. For a friend function declaration that is not a template declaration:

1. if the name of the friend is a qualified or unqualified template-id, the friend declaration refers to a specialization of a function template, otherwise,

2. if the name of the friend is a qualified-id and a matching non-template function is found in the specified class or namespace, the friend declaration refers to that function, otherwise,

3. if the name of the friend is a qualified-id and a matching function template is found in the specified class or namespace, the friend declaration refers to the deduced specialization of that function template (13.10.3.7), otherwise,

4. the name shall be an unqualified-id that declares (or redeclares) a non-template function.

[Example 1:]

```cpp
template<class T> class task;
template<class T> task<T>* preempt(task<T>*);

template<class T> class task {
        friend void next_time();
        friend void process(task<T>*);
        friend task<T>* preempt<T>(task<T>*);
        template<class C> friend int func(C);

        friend class task<int>;
        template<class P> friend class frd;
    };
```

Here, each specialization of the `task` class template has the function `next_time` as a friend; because `process` does not have explicit template-arguments, each specialization of the `task` class template has an appropriately typed function `process` as a friend, and this friend is not a function template specialization; because the friend `preempt` has an explicit template-argument `T`, each specialization of the `task` class template has the appropriate specialization of the function template `preempt` as a friend; and each specialization of the `task` class template has all specializations of the function template `func` as friends. Similarly, each specialization of the `task` class template has the class template specialization `task<int>` as a friend, and has all specializations of the class template `frd` as friends. —end example]

2 A friend template may be declared within a class or class template. A friend function template may be defined within a class or class template, but a friend class template may not be defined in a class or class template. In these cases, all specializations of the friend class or friend function template are friends of the class or class template granting friendship.

[Example 2:]

```cpp
class A {
        template<class T> friend class B; // OK
        template<class T> friend void f(T){ /* ... */ } // OK
    };
```

—end example]

A template friend declaration specifies that all specializations of that template, whether they are implicitly instantiated (13.9.2), partially specialized (13.7.6) or explicitly specialized (13.9.4), are friends of the class containing the template friend declaration.

[Example 3:]

```cpp
class X {
        template<class T> friend struct A;
        class Y { };
    };

template<class T> struct A { X::*Y ab; };
```

—end example]

§ 13.7.5
A template friend declaration may declare a member of a dependent type to be a friend. The friend declaration shall declare a function or specify a type with an elaborated-type-specifier, in either case with a nested-name-specifier ending with a simple-template-id, C, whose template-name names a class template. The template parameters of the template friend declaration shall be deducible from C (13.10.3.6). In this case, a member of a specialization S of the class template is a friend of the class granting friendship if deduction of the template parameters of C from S succeeds, and substituting the deduced template arguments into the friend declaration produces a declaration that would be a valid redeclaration of the member of the specialization.

[Example 4]:

```cpp
template<class T> struct A {
    struct B { }
    void f();
    struct D {
        void g();
    }
    T h();
    template<T U> T i();
};
template<> struct A<int> {
    struct B { }
    int f();
    struct D {
        void g();
    }
    template<int U> int i();
};
template<> struct A<float*> {
    int *h();
};
class C {
    template<class T> friend struct A<T>::B; // grants friendship to A<int>::B even though it is not a specialization of A<T>::B
    template<class T> friend void A<T>::f(); // does not grant friendship to A<int>::f()
    template<class T> friend void A<T>::D::g(); // because its return type does not match
    template<class T> friend int *A<T>::h(); // grants friendship to A<int>::h() and A<float>::h()
    template<class T> template<T U> friend T A<T>::i(); // grants friendship to instantiations of A<T>::i() and friend T A<T>::i(), and thereby to all specializations of those function templates
};
```

[Note 1]: A friend declaration can first declare a member of an enclosing namespace scope (13.8.6). — end note]

A friend template shall not be declared in a local class.

Friend declarations shall not declare partial specializations.

[Example 5]:

```cpp
template<class T> class A { }
class X {
    template<class T> friend class A<T>*; // error
};
```

When a friend declaration refers to a specialization of a function template, the function parameter declarations shall not include default arguments, nor shall the inline, constexpr, or consteval specifiers be used in such a declaration.

A non-template friend declaration with a requires-clause shall be a definition. A friend function template with a constraint that depends on a template parameter from an enclosing template shall be a definition. Such a constrained friend function or function template declaration does not declare the same function or function template as a declaration in any other scope.
13.7.6 Class template partial specializations

13.7.6.1 General

A primary class template declaration is one in which the class template name is an identifier. A template declaration in which the class template name is a simple-template-id is a partial specialization of the class template named in the simple-template-id. A partial specialization of a class template provides an alternative definition of the template that is used instead of the primary definition when the arguments in a specialization match those given in the partial specialization (13.7.6.2). The primary template shall be declared before any specializations of that template. A partial specialization shall be declared before the first use of a class template specialization that would make use of the partial specialization as the result of an implicit or explicit instantiation in every translation unit in which such a use occurs; no diagnostic is required.

2 Each class template partial specialization is a distinct template and definitions shall be provided for the members of a template partial specialization (13.7.6.4).

3 [Example 1]:

```cpp
template<class T1, class T2, int I> class A { }
template<class T, int I> class A<T, T*, I> { };
template<class T1, class T2, int I> class A<T1*, T2, I> { };
template<class T> class A<int, T*, 5> { };
template<class T1, class T2, int I> class A<T1, T2*, I> { };
```

The first declaration declares the primary (unspecialized) class template. The second and subsequent declarations declare partial specializations of the primary template. —end example

4 A class template partial specialization may be constrained (13.1).

[Example 2]:

```cpp
template<typename T> concept C = true;
template<typename T> struct X { };
template<typename T> struct X<T*> { }; // #1
template<C T> struct X<T> { }; // #2
```

Both partial specializations are more specialized than the primary template. #1 is more specialized because the deduction of its template arguments from the template argument list of the class template specialization succeeds, while the reverse does not. #2 is more specialized because the template arguments are equivalent, but the partial specialization is more constrained (13.5.5). —end example

5 The template parameters are specified in the angle bracket enclosed list that immediately follows the keyword template. For partial specializations, the template argument list is explicitly written immediately following the class template name. For primary templates, this list is implicitly described by the template parameter list. Specifically, the order of the template arguments is the sequence in which they appear in the template parameter list.

[Example 3: The template argument list for the primary template in the example above is <T1, T2, I>. —end example]

[Note 1: The template argument list cannot be specified in the primary template declaration. For example,]

```cpp
template<class T1, class T2, int I>
class A<T1, T2, I> { }; // error
```

—end note

6 A class template partial specialization may be declared in any scope in which the corresponding primary template may be defined (9.8.2.3, 11.4, 13.7.3).

[Example 4:]

```cpp
template<class T> struct A {
    struct C {
        template<class T2> struct B { };
        template<class T2> struct B<T2**> { }; // partial specialization #1
    };

    // partial specialization of A<T>::C::B<T2>
    template<class T> template<class T2>
        struct A<T>::C::B<T2> { }; // #2
```
Partial specialization declarations themselves are not found by name lookup. Rather, when the primary template name is used, any previously-declared partial specializations of the primary template are also considered. One consequence is that a using-declaration which refers to a class template does not restrict the set of partial specializations which may be found through the using-declaration.

Example 5:

```cpp
namespace N {
    template<class T1, class T2> class A { }; // primary template
}
using N::A; // refers to the primary template

namespace N {
    template<class T> class A<T, T*> { }; // partial specialization
}

A<int,int*> a; // uses the partial specialization, which is found through the using-declaration
                 // which refers to the primary template
```

A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

Within the argument list of a class template partial specialization, the following restrictions apply:

1. The type of a template parameter corresponding to a specialized non-type argument shall not be dependent on a parameter of the specialization.

Example 6:

```cpp
template <class T, T t> struct C {}; // error

template< int X, int (*array_ptr)[X] > class A {}; // error
```

2. The specialization shall be more specialized than the primary template (13.7.6.3).

3. The template parameter list of a specialization shall not contain default template argument values.

4. An argument shall not contain an unexpanded pack. If an argument is a pack expansion (13.7.4), it shall be the last argument in the template argument list.

The usual access checking rules do not apply to non-dependent names used to specify template arguments of the simple-template-id of the partial specialization.

Note 2: The template arguments can be private types or objects that would normally not be accessible. Dependent names cannot be checked when declaring the partial specialization, but will be checked when substituting into the partial specialization. —end note

### 13.7.6.2 Matching of class template partial specializations [temp.class.spec.match]

When a class template is used in a context that requires an instantiation of the class, it is necessary to determine whether the instantiation is to be generated using the primary template or one of the partial specializations. This is done by matching the template arguments of the class template specialization with the template argument lists of the partial specializations.

1. If exactly one matching specialization is found, the instantiation is generated from that specialization.
2. If more than one matching specialization is found, the partial order rules (13.7.6.3) are used to determine whether one of the specializations is more specialized than the others. If none of the specializations is more specialized than all of the other matching specializations, then the use of the class template is ambiguous and the program is ill-formed.

---

138 There is no way in which they could be used.
If no matches are found, the instantiation is generated from the primary template.

A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (13.10.3), and the deduced template arguments satisfy the associated constraints of the partial specialization, if any (13.5.3).

Example 1:

```cpp
#include <iostream>

int main() {
    A<int, int, 1> a1; // uses #1
    A<int, int*, 1> a2; // uses #2, T is int, I is 1
    A<int, char*, 5> a3; // uses #4, T is char
    A<int, char*, 1> a4; // uses #5, T1 is int, T2 is char, I is 1
    A<int*, int*, 2> a5; // ambiguous: matches #3 and #5
    return 0;
}
```

Example 2:

```cpp
#include <iostream>

int main() {
    S<int> s1; // uses #1; the constraints of #2 are not satisfied
    S<Arg> s2; // uses #2; both constraints are satisfied but #2 is more specialized
    return 0;
}
```

If the template arguments of a partial specialization cannot be deduced because of the structure of its `template-parameter-list` and the `template-id`, the program is ill-formed.

Example 3:

```cpp
#include <iostream>

int main() {
    A<int, int, 1> a1; // uses #1
    A<int, int*, 1> a2; // uses #2, T is int, I is 1
    A<int, char*, 5> a3; // uses #4, T is char
    A<int, char*, 1> a4; // uses #5, T1 is int, T2 is char, I is 1
    A<int*, int*, 2> a5; // ambiguous: matches #3 and #5
    return 0;
}
```

In a type name that refers to a class template specialization, (e.g., `A<int, int, 1>`) the argument list shall match the template parameter list of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template.

13.7.6.3 Partial ordering of class template specializations

For two class template partial specializations, the first is more specialized than the second if, given the following rewrite to two function templates, the first function template is more specialized than the second according to the ordering rules for function templates (13.7.7.3):

1. Each of the two function templates has the same template parameters and associated constraints (13.5.3) as the corresponding partial specialization.

2. Each function template has a single function parameter whose type is a class template specialization where the template arguments are the corresponding template parameters from the function template for each template argument in the `template-argument-list` of the `simple-template-id` of the partial specialization.

Example 1:

```cpp
#include <iostream>

int main() {
    X<int, int, 1> x1; // uses #1
    X<int, int*, 1> x2; // uses #2, T is int, I is 1
    X<int, char*, 5> x3; // uses #4, T is char
    X<int, char*, 1> x4; // uses #5, T1 is int, T2 is char, I is 1
    X<int*, int*, 2> x5; // ambiguous: matches #3 and #5
    return 0;
}
```
According to the ordering rules for function templates, the function template \( B \) is more specialized than the function template \( A \) and the function template \( D \) is more specialized than the function template \( C \). Therefore, the partial specialization \#2 is more specialized than the partial specialization \#1 and the partial specialization \#4 is more specialized than the partial specialization \#3. — end example

Example 2:

\[
\begin{align*}
    &\text{template<typename T> concept C = requires (T t) { t.f(); };} \\
    &\text{template<typename T> concept D = C<T> && requires (T t) { t.f(); };} \\
    &\text{template<typename T> class S { };} \\
    &\text{template<typename T> class S<T> { };} \\
    &\text{template<typename T> concept C T> class S<T> { };} \\
    &\text{template<C T> class S<T> { };} \\
    &\text{template<C T> void f(S<T>);} \\
    &\text{template<D T> void f(S<T>);}
\end{align*}
\]

The partial specialization \#2 is more specialized than \#1 because \( B \) is more specialized than \( A \). — end example

13.7.6.4 Members of class template specializations

The template parameter list of a member of a class template partial specialization shall match the template parameter list of the class template partial specialization. The template argument list of a member of a class template partial specialization shall match the template argument list of the class template partial specialization. A class template partial specialization is a distinct template. The members of the class template partial specialization are unrelated to the members of the primary template. Class template partial specialization members that are used in a way that requires a definition shall be defined; the definitions of members of the primary template are never used as definitions for members of a class template partial specialization. An explicit specialization of a member of a class template partial specialization is declared in the same way as an explicit specialization of the primary template.

Example 1:

\[
\begin{align*}
    &\text{primary class template} \\
    &\text{template<class T, int I> struct A { void f(); };} \\
    &\text{member of primary class template} \\
    &\text{template<class T, int I> void A<T, I>::f() \{} \} \}
\end{align*}
\]

\[
\begin{align*}
    &\text{class template partial specialization} \\
    &\text{template<class T> struct A<T,2> { void f(); void g(); void h(); };} \\
    &\text{member of class template partial specialization} \\
    &\text{template<class T> void A<T,2>::f() \{} \} \\
    &\text{explicit specialization} \\
    &\text{template<> void A<char,2>::h() \{} \}
\end{align*}
\]
int main() {
    A<char,0> a0;
    A<char,2> a2;

    a0.f(); // OK, uses definition of primary template's member
    a2.g(); // OK, uses definition of partial specialization's member
    a2.h(); // OK, uses definition of explicit specialization's member
    a2.f(); // error: no definition of f for A<T,2>; the primary template is not used here
}

—end example]  

If a member template of a class template is partially specialized, the member template partial specializations are member templates of the enclosing class template; if the enclosing class template is instantiated (13.9.2, 13.9.3), a declaration for every member template partial specialization is also instantiated as part of creating the members of the class template specialization. If the primary member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the partial specializations of the member template are ignored for this specialization of the enclosing class template. If a partial specialization of the member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the primary member template and its other partial specializations are still considered for this specialization of the enclosing class template.

Example 2:

```cpp
template<class T> struct A {
    template<class T2> struct B {};
    template<class T2> struct B<T2*> {};
};

template<> template<class T2> struct A<short>::B {};

A<char>::B<int*> abcip; // uses #2
A<short>::B<int*> absip; // uses #3
A<char>::B<int> abci; // uses #1
```

—end example]

### 13.7.7 Function templates [temp.fct]

#### 13.7.7.1 General [temp.fct.general]

A function template defines an unbounded set of related functions.

Example 1: A family of sort functions might be declared like this:

```cpp
template<class T> class Array { }
template<class T> void sort(Array<T>&);
```

—end example]

2 A function template can be overloaded with other function templates and with non-template functions (9.3.4.6). A non-template function is not related to a function template (i.e., it is never considered to be a specialization), even if it has the same name and type as a potentially generated function template specialization.

13.7.7.2 Function template overloading [temp.over.link]

It is possible to overload function templates so that two different function template specializations have the same type.

Example 1:

```cpp
// translation unit 1:
// translation unit 2:
template<class T>
template<class T>
void f(T*);
void f(T*);
void g(int* p) {
    f(p); // calls f<int>(int*)
    f(p); // calls f<int*>(int*)
}
```

—end example]

139) That is, declarations of non-template functions do not merely guide overload resolution of function template specializations with the same name. If such a non-template function is odr-used (6.3) in a program, it must be defined; it will not be implicitly instantiated using the function template definition.
Such specializations are distinct functions and do not violate the one-definition rule (6.3).

The signature of a function template is defined in Clause 3. The names of the template parameters are significant only for establishing the relationship between the template parameters and the rest of the signature. [Note 1: Two distinct function templates can have identical function return types and function parameter lists, even if overload resolution alone cannot distinguish them.]

```cpp
template<class T> void f();
template<int I> void f();      // OK: overloads the first template
                                // distinguishable with an explicit template argument list
```

When an expression that references a template parameter is used in the function parameter list or the return type in the declaration of a function template, the expression that references the template parameter is part of the signature of the function template. This is necessary to permit a declaration of a function template in one translation unit to be linked with another declaration of the function template in another translation unit and, conversely, to ensure that function templates that are intended to be distinct are not linked with one another. [Note 1: Two distinct function templates can have identical function return types and function parameter lists, even if overload resolution alone cannot distinguish them.]

```cpp
template<int I, int J> A<I+J> f(A<I>, A<J>); // #1
template<int K, int L> A<K+L> f(A<K>, A<L>); // same as #1
template<int I, int J> A<I-J> f(A<I>, A<J>); // different from #1
```

Two expressions involving template parameters are considered equivalent if two function definitions containing the expressions would satisfy the one-definition rule (6.3), except that the tokens used to name the template parameters may differ as long as a token used to name a template parameter in one expression is replaced by another token that names the same template parameter in the other expression. Two unevaluated operands that do not involve template parameters are considered equivalent if two function definitions containing the expressions would satisfy the one-definition rule, except that the tokens used to name types and declarations may differ as long as they name the same entities, and the tokens used to form concept-ids may differ as long as the two template-ids are the same (13.6).

[Note 3: For instance, A<42> and A<40+2> name the same type. — end note]

Two lambda-expressions are never considered equivalent. [Note 4: The intent is to avoid lambda-expressions appearing in the signature of a function template with external linkage. — end note]

For determining whether two dependent names (13.8.3) are equivalent, only the name itself is considered, not the result of name lookup in the context of the template. If multiple declarations of the same function template differ in the result of this name lookup, the result for the first declaration is used.

[Example 3:]

```cpp
template <int I, int J> void f(A<I+J>);       // #1
template <int K, int L> void f(A<K+L>);       // same as #1

template <class T> decltype(g(T())) h();
int g(int);
template <class T> decltype(g(T())) h()       // redeclaration of h() uses the earlier lookup...
{ return g(T()); }
int i = h<int>();                                // ... although the lookup here does find g(int)
// template argument substitution fails; g(int)
// was not in scope at the first declaration of h()

// ill-formed, no diagnostic required: the two expressions are functionally equivalent but not equivalent
template <int N> void foo(const char (*s)[(N)]);
template <int N> void foo(const char (*s)[(N)]);

// two different declarations because the non-dependent portions are not considered equivalent
template <class T> void spam(decltype((T)[])(*s)[sizeof(T)]);
template <class T> void spam(decltype((T)[])(*s)[sizeof(T)]);
```

§ 13.7.7.2 396
Two potentially-evaluated expressions involving template parameters that are not equivalent are *functionally equivalent* if, for any given set of template arguments, the evaluation of the expression results in the same value. Two unevaluated operands that are not equivalent are functionally equivalent if, for any given set of template arguments, the expressions perform the same operations in the same order with the same entities.

[Note 5: For instance, one could have redundant parentheses. — end note]

Two template-heads are *equivalent* if their template-parameter-lists have the same length, corresponding template-parameters are equivalent and are both declared with type-constraints that are equivalent if either template-parameter is declared with a type-constraint, and if either template-head has a requires-clause, they both have requires-clauses and the corresponding constraint-expressions are equivalent. Two template-parameters are equivalent under the following conditions:

1. they declare template parameters of the same kind,
2. if either declares a template parameter pack, they both do,
3. if they declare non-type template parameters, they have equivalent types ignoring the use of type-constraints for placeholder types, and
4. if they declare template template parameters, their template parameters are equivalent.

When determining whether types or type-constraints are equivalent, the rules above are used to compare expressions involving template parameters. Two template-heads are functionally equivalent if they accept and are satisfied by (13.5.2) the same set of template argument lists.

Two function templates are equivalent if they are declared in the same scope, have the same name, have equivalent template-heads, and have return types, parameter lists, and trailing requires-clauses (if any) that are equivalent using the rules described above to compare expressions involving template parameters. Two function templates are functionally equivalent if they are declared in the same scope, have the same name, accept and are satisfied by the same set of template argument lists, and have return types and parameter lists that are functionally equivalent using the rules described above to compare expressions involving template parameters. If the validity or meaning of the program depends on whether two constructs are equivalent, and they are functionally equivalent but not equivalent, the program is ill-formed, no diagnostic required.

[Note 6: This rule guarantees that equivalent declarations will be linked with one another, while not requiring implementations to use heroic efforts to guarantee that functionally equivalent declarations will be treated as distinct. For example, the last two declarations are functionally equivalent and would cause a program to be ill-formed:

```c++
// guaranteed to be the same
template <int I> void f(A<I>, A<I+10>);

// guaranteed to be different
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+11>);

// ill-formed, no diagnostic required
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+1+2+3+4>);
— end note]

13.7.7.3 Partial ordering of function templates

If a function template is overloaded, the use of a function template specialization can be ambiguous because template argument deduction (13.10.3) may associate the function template specialization with more than one function template declaration. Partial ordering of overloaded function template declarations is used in the following contexts to select the function template to which a function template specialization refers:

1. during overload resolution for a call to a function template specialization (12.4.4);
2. when the address of a function template specialization is taken;
3. when a placement operator delete that is a function template specialization is selected to match a placement operator new (6.7.5.5.3, 7.6.2.8);
4. when a friend function declaration (13.7.5), an explicit instantiation (13.9.3) or an explicit specialization (13.9.4) refers to a function template specialization.
Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function type. The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process. If both deductions succeed, the partial ordering selects the more constrained template (if one exists) as determined below.

To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs (13.7.4) thereof) synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template.

[Note 1: The type replacing the placeholder in the type of the value synthesized for a non-type template parameter is also a unique synthesized type. — end note]

Each function template \( M \) that is a member function is considered to have a new first parameter of type \( X(M) \), described below, inserted in its function parameter list. If exactly one of the function templates was considered by overload resolution via a rewritten candidate (12.4.2.3) with a reversed order of parameters, then the order of the function parameters in its transformed template is reversed. For a function template \( M \) with cv-qualifiers \( \text{cv} \) that is a member of a class \( A \):

1. The type \( X(M) \) is “rvalue reference to \( \text{cv} \ A \)” if the optional \( \text{ref-qualifier} \) of \( M \) is \&\& or if \( M \) has no \( \text{ref-qualifier} \) and the positionally-corresponding parameter of the other transformed template has rvalue reference type; if this determination depends recursively upon whether \( X(M) \) is an rvalue reference type, it is not considered to have rvalue reference type.

2. Otherwise, \( X(M) \) is “lvalue reference to \( \text{cv} \ A \)”.

[Note 2: This allows a non-static member to be ordered with respect to a non-member function and for the results to be equivalent to the ordering of two equivalent non-members. — end note]

[Example 1:]

```cpp
struct A { }
template<class T> struct B {
    template<class R> int operator*(R&);
    // #1
};

template<class T, class R> int operator*(T&, R&);
// #2

// The declaration of B::operator* is transformed into the equivalent of
// template<class R> int operator*(B<A>&, R&);
// #1a

int main() {
    A a;
    B<A> b;
    b * a; // calls #1
}

— end example]

Using the transformed function template’s function type, perform type deduction against the other template as described in 13.10.3.5.

[Example 2:]

```cpp
template<class T> struct A { A(); }

template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);

template<class T> void g(T);
template<class T> void g(T&);

template<class T> void h(const T&);
template<class T> void h(A<T>&);

void m() {
    const int* p;
    f(p); // f(const T*) is more specialized than f(T) or f(T*)
}

§ 13.7.7.3 398
float x;
g(x); // ambiguous: g(T) or g(Tk)
A<int> z;
h(z); // overload resolution selects h(A<T>&)
const A<int> z2;
h(z2); // h(const Tk) is called because h(A<T>&) is not callable
}

— end example

[Note 3: Since, in a call context, such type deduction considers only parameters for which there are explicit call arguments, some parameters are ignored (namely, function parameter packs, parameters with default arguments, and ellipsis parameters).

Example 3:

```cpp
template<class T> void f(T);
// #1
template<class T> void f(T*, int=1); // #2
template<class T> void g(T); // #3
template<class T> void g(T*, ...); // #4
int main() {
  int* ip;
f(ip); // calls #2
g(ip); // calls #4
}
— end example
```

Example 4:

```cpp
template<class T, class U> struct A { }

template<class T, class U> void f(U, A<U, T>* p = 0); // #1
template<class T> void f(U, A<T, U>* p = 0); // #2
template<class T> void g(T, T = T()); // #3
template<class T, class... U> void g(T, U...); // #4
void h() {
  f<int>(42, (A<int, int>*)0); // calls #2
  f<int>(42); // error: ambiguous
  g(42); // error: ambiguous
}
— end example
```

Example 5:

```cpp
template<class T, class... U> void f(T, U...); // #1
template<class T> void f(T); // #2
template<class T, class... U> void g(T*, U...); // #3
template<class T> void g(T); // #4
void h(int i) {
  f(&i); // OK: calls #2
  g(&i); // OK: calls #3
}
— end example
— end note
```

6 If deduction against the other template succeeds for both transformed templates, constraints can be considered as follows:

(6.1) — If their template-parameter-lists (possibly including template-parameters invented for an abbreviated function template (9.3.4.6)) or function parameter lists differ in length, neither template is more specialized than the other.

(6.2) — Otherwise:

(6.2.1) — If exactly one of the templates was considered by overload resolution via a rewritten candidate with reversed order of parameters:
If, for either template, some of the template parameters are not deducible from their function parameters, neither template is more specialized than the other.

If there is either no reordering or more than one reordering of the associated template-parameter-list such that:

- the corresponding template-parameters of the template-parameter-lists are equivalent and
- the function parameters that positionally correspond between the two templates are of the same type,

neither template is more specialized than the other.

Otherwise, if the corresponding template-parameters of the template-parameter-lists are not equivalent (13.7.7.2) or if the function parameters that positionally correspond between the two templates are not of the same type, neither template is more specialized than the other.

Otherwise, if the context in which the partial ordering is done is that of a call to a conversion function and the return types of the templates are not the same, then neither template is more specialized than the other.

Otherwise, if one template is more constrained than the other (13.5.5), the more constrained template is more specialized than the other.

Otherwise, neither template is more specialized than the other.

Example 6:

```cpp
template <typename> constexpr bool True = true;
template <typename T> concept C = True<T>;
void f(C auto &, auto &) = delete;
template <C Q> void f(Q &, C auto &);

void g(struct A *ap, struct B *bp) {
    f(*ap, *bp);    // OK: Can use different methods to produce template parameters
}

template <typename T, typename U> struct X {};
template <typename T, C U, typename V> bool operator==(X<T, U>, V) = delete;
template <C T, C U, C V> bool operator==(T, X<U, V>);

void h() {
    X<int *, int>{} == 0;    // OK: Correspondence of [T, U, V] and [U, V, T]
}
```

13.7.8 Alias templates

A template-declaration in which the declaration is an alias-declaration (9.1) declares the identifier to be an alias template. An alias template is a name for a family of types. The name of the alias template is a template-name.

When a template-id refers to the specialization of an alias template, it is equivalent to the associated type obtained by substitution of its template-arguments for the template-parameters in the defining-type-id of the alias template.

Example 1:

```cpp
template<class T> struct Alloc { /* ... */ };
template<class T> using Vec = vector<T, Alloc<T>>;
Vec<int> v;   // same as vector<int, Alloc<int>> v;

template<class T>
void process(Vec<T>& v)
{ /* ... */ }
```
template<class T>
void process(vector<T, Alloc<T>>& w)
{ /* ... */ } // error: redefinition

template<template<class> class TT>
void f(TT<int>);
f(v); // error: Vec not deduced

template<template<class,class> class TT>
void g(TT<int, Alloc<int>>);
g(v); // OK: TT = vector
—end example]

However, if the template-id is dependent, subsequent template argument substitution still applies to the template-id.

[Example 2:

template<typename...> using void_t = void;
template<typename T> void_t<typename T::foo> f();
f<int>(); // error: int does not have a nested type foo
—end example]

The defining-type-id in an alias template declaration shall not refer to the alias template being declared. The type produced by an alias template specialization shall not directly or indirectly make use of that specialization.

[Example 3:

template <class T> struct A;
template <class T> using B = typename A<T>::U;
template <class T> struct A {
  typedef B<T> U;
};
B<short> b; // error: instantiation of B<short> uses own type via A<short>::U
—end example]

The type of a lambda-expression appearing in an alias template declaration is different between instantiations of that template, even when the lambda-expression is not dependent.

[Example 4:

template <class T>
using A = decltype([] {}); // A<int> and A<char> refer to different closure types
—end example]

13.7.9 Concept definitions [temp.concept]

A concept is a template that defines constraints on its template arguments.

concept-definition:
  concept concept-name = constraint-expression ;

concept-name:
  identifier

A concept-definition declares a concept. Its identifier becomes a concept-name referring to that concept within its scope.

[Example 1:

template<typename T>
concept C = requires(T x) {
  { x == x } -> std::convertible_to<bool>;
};

template<typename T>
requires C<T> // C constrains f1(T) in constraint-expression
T f1(T x) { return x; }

§ 13.7.9
template<C T> // C, as a type-constraint, constrains f2(T)
T f2(T x) { return x; }
—end example]

3 A concept-definition shall appear at namespace scope (6.4.6).

4 A concept shall not have associated constraints (13.5.3).

5 A concept is not instantiated (13.9).

[Note 1: A concept-id (13.3) is evaluated as an expression. A concept cannot be explicitly instantiated (13.9.3), explicitly specialized (13.9.4), or partially specialized. — end note]

6 The constraint-expression of a concept-definition is an unevaluated operand (7.2.3).

7 The first declared template parameter of a concept definition is its prototype parameter. A type concept is a concept whose prototype parameter is a type template-parameter.

13.8 Name resolution

13.8.1 General

Three kinds of names can be used within a template definition:
1. The name of the template itself, and names declared within the template itself.
2. Names dependent on a template-parameter (13.8.3).
3. Names from scopes which are visible within the template definition.

2 A name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword typename.

[Example 1:

// no B declared here
class X;
template<class T> class Y {
  class Z;
  // forward declaration of member class
  void f() {
    X* a1; // declare pointer to X
    T* a2; // declare pointer to T
    Y* a3; // declare pointer to Y<T>
    Z* a4; // declare pointer to Z
    typedef typename T::A TA;
    TA* a5; // declare pointer to T's A
    typename T::A* a6; // declare pointer to T's A
    T::A* a7; // error: no visible declaration of a7
    // T::A is not a type name; multiplication of T::A by a7
    B* a8; // error: no visible declarations of B and a8
    // B is not a type name; multiplication of B by a8
  }
};
—end example]

typename-specifier:
typename nested-name-specifier identifier
typename nested-name-specifier template_opt simple-template-id

3 A typename-specifier denotes the type or class template denoted by the simple-type-specifier (9.2.9.3) formed by omitting the keyword typename. The usual qualified name lookup (6.5.4) is used to find the qualified-id even in the presence of typename.

[Example 2:

struct A {
  struct X { }; // forward declaration of member class
  int X;
};

§ 13.8.1
struct B {
    struct X { }
};
template<class T> void f(T t) {
    typename T::X x;
}
void foo() {
    A a;
    B b;
    f(b);       // OK: T::X refers to B::X
    f(a);       // error: T::X refers to the data member A::X not the struct A::X
}
— end example

4 A qualified name used as the name in a class-or-decltype (11.7) or an elaborated-type-specifier is implicitly assumed to name a type, without the use of the typename keyword. In a nested-name-specifier that immediately contains a nested-name-specifier that depends on a template parameter, the identifier or simple-template-id is implicitly assumed to name a type, without the use of the typename keyword.

[Note 1: The typename keyword is not permitted by the syntax of these constructs. — end note]

5 A qualified-id is assumed to name a type if

(5.1) — it is a qualified name in a type-id-only context (see below), or
(5.2) — it is a decl-specifier of the decl-specifier-seq of a

(5.2.1) — simple-declaration or a function-definition in namespace scope,
(5.2.2) — member-declaration,
(5.2.3) — parameter-declaration in a member-declaration\(^\text{140}\), unless that parameter-declaration appears in a default argument,
(5.2.4) — parameter-declaration in a declarator of a function or function template declaration whose declarator-id is qualified, unless that parameter-declaration appears in a default argument,
(5.2.5) — parameter-declaration in a lambda-declarator or requirement-parameter-list, unless that parameter-declaration appears in a default argument, or
(5.2.6) — parameter-declaration of a (non-type) template-parameter.

A qualified name is said to be in a type-id-only context if it appears in a type-id, new-type-id, or defining-type-id and the smallest enclosing type-id, new-type-id, or defining-type-id is a new-type-id, defining-type-id, trailing-return-type, default argument of a type-parameter of a template, or type-id of a static_cast, const_cast, reinterpret_cast, or dynamic_cast.

[Example 3:

```
template<class T> T::R f();       // OK, return type of a function declaration at global scope
template<class T> void f(T::R);   // ill-formed, no diagnostic required: attempt to declare
// a void variable template

template<class T> struct S {
    using Ptr = PtrTraits<T>::Ptr;
    T::R f(T::P p) {         // OK, in a defining-type-id
        return static_cast<T::R>(p);
    }
    auto g() -> S<T*>::Ptr;  // OK, trailing-return-type
};
template<typename T> void f() {         // variable pf of type void* initialized with T::X
    void (*pf)(T::X);
    void g(T::X);            // error: T::X at block scope does not denote a type
// (attempt to declare a void variable)
}
— end example]

6 A qualified-id that refers to a member of an unknown specialization, that is not prefixed by typename, and that is not otherwise assumed to name a type (see above) denotes a non-type.

\(^{140}\) This includes friend function declarations.
Example 4:
```c
template <class T> void f(int i) {
    T::x * i;  // expression, not the declaration of a variable i
}

struct Foo {
    typedef int x;
};

struct Bar {
    static int const x = 5;
};

int main() {
    f<Bar>(1);  // OK
    f<Foo>(1);  // error: Foo::x is a type
}
```

Within the definition of a class template or within the definition of a member of a class template following the `declarator-id`, the keyword `typename` is not required when referring to a member of the current instantiation (13.8.3.2).

Example 5:
```c
template<class T> struct A {
    typedef int B;
    B b;  // OK, no typename required
};
```

The validity of a template may be checked prior to any instantiation.

Note 2: Knowing which names are type names allows the syntax of every template to be checked in this way. — end note

The program is ill-formed, no diagnostic required, if:

(8.1) — no valid specialization can be generated for a template or a substatement of a constexpr if statement (8.5.2) within a template and the template is not instantiated, or

(8.2) — no substitution of template arguments into a `type-constraint` or `requires-clause` would result in a valid expression, or

(8.3) — every valid specialization of a variadic template requires an empty template parameter pack, or

(8.4) — a hypothetical instantiation of a template immediately following its definition would be ill-formed due to a construct that does not depend on a template parameter, or

(8.5) — the interpretation of such a construct in the hypothetical instantiation is different from the interpretation of the corresponding construct in any actual instantiation of the template.

Note 3: This can happen in situations including the following:

(8.5.1) — a type used in a non-dependent name is incomplete at the point at which a template is defined but is complete at the point at which an instantiation is performed, or

(8.5.2) — lookup for a name in the template definition found a `using-declaration`, but the lookup in the corresponding scope in the instantiation does not find any declarations because the `using-declaration` was a pack expansion and the corresponding pack is empty, or

(8.5.3) — an instantiation uses a default argument or default template argument that had not been defined at the point at which the template was defined, or

(8.5.4) — constant expression evaluation (7.7) within the template instantiation uses

(8.5.4.1) — the value of a const object of integral or unscoped enumeration type or

(8.5.4.2) — the value of a `constexpr` object or

(8.5.4.3) — the value of a reference or

(8.5.4.4) — the definition of a constexpr function,

§ 13.8.1
and that entity was not defined when the template was defined, or

(8.5.5) a class template specialization or variable template specialization that is specified by a non-dependent

simple-template-id is used by the template, and either it is instantiated from a partial specialization that

was not defined when the template was defined or it names an explicit specialization that was not declared

when the template was defined.

— end note

Otherwise, no diagnostic shall be issued for a template for which a valid specialization can be generated.

[Note 4: If a template is instantiated, errors will be diagnosed according to the other rules in this document. Exactly

when these errors are diagnosed is a quality of implementation issue. — end note]

[Example 6:

```cpp
int j;
template<T> class X {
    void f(T t, int i, char* p) {
        t = i; // diagnosed if X::f is instantiated, and the assignment to t is an error
        p = i; // may be diagnosed even if X::f is not instantiated
        p = j; // may be diagnosed even if X::f is not instantiated
    }
    void g(T t) {
        // may be diagnosed even if X::g is not instantiated
    }
};
template<class... T> struct A {
    void operator++(int, T... t); // error: too many parameters
};
template<class... T> union X : T... { }; // error: union with base class
template<class... T> struct A : T..., T... { }; // error: duplicate base class
— end example
```

] 9 When looking for the declaration of a name used in a template definition, the usual lookup rules (6.5.2, 6.5.3)

are used for non-dependent names. The lookup of names dependent on the template parameters is postponed

until the actual template-argument is known (13.8.3).

[Example 7:

```cpp
#include <iostream>
using namespace std;
template<T> class Set {
    T* p;
    int cnt;
public:
    Set();
    Set(const Set&) ;
    void printall() {
        for (int i = 0; i<cnt; i++)
            cout << p[i] << '\n';
    }
};
```

In the example, i is the local variable i declared in printall, cnt is the member cnt declared in Set, and cout is the

standard output stream declared in iostream. However, not every declaration can be found this way; the resolution

of some names is postponed until the actual template-arguments are known. For example, even though the name

operator<< is known within the definition of printall() and a declaration of it can be found in <iostream>, the

actual declaration of operator<< needed to print p[i] cannot be known until it is known what type T is (13.8.3).

— end example]

] 10 If a name does not depend on a template-parameter (as defined in 13.8.3), a declaration (or set of declarations) for

that name shall be in scope at the point where the name appears in the template definition; the name is

bound to the declaration (or declarations) found at that point and this binding is not affected by declarations that

are visible at the point of instantiation.

[Example 8:

```cpp
void f(char);
```
template<class T> void g(T t) {
    f(1);    // f(char)
    f(T(1));   // dependent
    f(t);     // dependent
    dd++;     // not dependent; error: declaration for dd not found
}

enum E { e }; void f(E);

double dd;
void h() {
    g(e);     // will cause one call of f(char) followed by two calls of f(E)
    g('a');  // will cause three calls of f(char)
}

—end example

11 [Note 5: For purposes of name lookup, default arguments and noexcept-specifiers of function templates and default arguments and noexcept-specifiers of member functions of class templates are considered definitions (13.7). —end note]

13.8.2 Locally declared names [temp.local]

1 Like normal (non-template) classes, class templates have an injected-class-name (11.1). The injected-class-name can be used as a template-name or a type-name. When it is used with a template-argument-list, as a template-argument for a template template-parameter, or as the final identifier in the elaborated-type-specifier of a friend class template declaration, it is a template-name that refers to the class template itself. Otherwise, it is a type-name equivalent to the template-name followed by the template-parameters of the class template enclosed in <>.

2 Within the scope of a class template specialization or partial specialization, when the injected-class-name is used as a type-name, it is equivalent to the template-name followed by the template-arguments of the class template specialization or partial specialization enclosed in <>.

[Example 1]:

    template<template<class> class T> class A { }
    template<class T> class Y;
    template<class T> class Y<int> {
        Y* p;           // meaning Y<int>
        Y<char>* q;    // meaning Y<char>
        A<Y>* a;       // meaning A::<Y>
        class B {
            template<class> friend class Y;  // meaning ::Y
        };
    };
—end example]

3 The injected-class-name of a class template or class template specialization can be used as either a template-name or a type-name wherever it is in scope.

[Example 2]:

    template <class T> struct Base { 
        Base* p;
    };

    template <class T> struct Derived: public Base<T> {
        typename Derived::Base* p;       // meaning Derived::Base<T>
    };

    template<class T, template<class> class U = T::template Base> class Third { 
    Third<Derived<int>> t;          // OK: default argument uses injected-class-name as a template
—end example]

4 A lookup that finds an injected-class-name (11.8) can result in an ambiguity in certain cases (for example, if it is found in more than one base class). If all of the injected-class-names that are found refer to specializations
of the same class template, and if the name is used as a *template-name*, the reference refers to the class

template itself and not a specialization thereof, and is not ambiguous.

**Example 3:**

```cpp
template <class T> struct Base { }

template <class T> struct Derived: Base<int>, Base<char> {
    typename Derived::Base b; // error: ambiguous
    typename Derived::Base<double> d; // OK
};

—end example
```

When the normal name of the template (i.e., the name from the enclosing scope, not the injected-class-name)
is used, it always refers to the class template itself and not a specialization of the template.

**Example 4:**

```cpp
template<class T> class X {
    X* p; // meaning X<T>
    X<int>* p2;
    X<int>* p3;
    ::X* p4; // error: missing template argument list
    // ::X does not refer to the injected-class-name
};

—end example
```

The name of a *template-parameter* shall not be redeclared within its scope (including nested scopes). A
*template-parameter* shall not have the same name as the template name.

**Example 5:**

```cpp
template<class T, int i> class Y {
    int T; // error: template-parameter redeclared
    void f() {
        char T; // error: template-parameter redeclared
    }
};

template<class X> class X;
// error: template-parameter redeclared

—end example
```

In the definition of a member of a class template that appears outside of the class template definition, the
name of a member of the class template hides the name of a *template-parameter* of any enclosing class
templates (but not a *template-parameter* of the member if the member is a class or function template).

**Example 6:**

```cpp
template<class T> struct A {
    struct B { /* ... */
    }
    typedef void C;
    void f();
    template<class U> void g(U);
};

template<class B> void A<B>::f() {
    B b; // A’s B, not the template parameter
}

template<class B> template<class C> void A<B>::g(C) {
    B b; // A’s B, not the template parameter
    C c; // the template parameter C, not A’s C
}

—end example
```

In the definition of a member of a class template that appears outside of the namespace containing the class
template definition, the name of a *template-parameter* hides the name of a member of this namespace.

**Example 7:**

§ 13.8.2
namespace N {
    class C { }
    template<class T> class B {
        void f(T);
    };
}

template<class C> void N::B<C>::f(C) {
    C b; // C is the template parameter, not N::C
}

In the definition of a class template or in the definition of a member of such a template that appears outside of the template definition, for each non-dependent base class (13.8.3.2), if the name of the base class or the name of a member of the base class is the same as the name of a template-parameter, the base class name or member name hides the template-parameter name (6.4.10).

[Example 8:
  struct A {
      struct B { /* ... */ }; // A's B
      int a;
      int Y;
  };

  template<class B, class a> struct X : A {
      B b; // A's a isn't a type name
      a b; // error: A's a isn't a type name
  }
—end example]

### 13.8.3 Dependent names

#### 13.8.3.1 General

Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct depends on the template parameters. In particular, types and expressions may depend on the type and/or value of template parameters (as determined by the template arguments) and this determines the context for name lookup for certain names. An expression may be type-dependent (that is, its type may depend on a template parameter) or value-dependent (that is, its value when evaluated as a constant expression (7.7) may depend on a template parameter) as described below.

1. In an expression of the form:

   \[
   \text{postfix-expression ( expression-list_{opt} )}
   \]

   where the postfix-expression is an unqualified-id, the unqualified-id denotes a dependent name if

   (2.1) any of the expressions in the expression-list is a pack expansion (13.7.4),

   (2.2) any of the expressions or braced-init-lists in the expression-list is type-dependent (13.8.3.3), or

   (2.3) the unqualified-id is a template-id in which any of the template arguments depends on a template parameter.

   If an operand of an operator is a type-dependent expression, the operator also denotes a dependent name.

   [Note 1: Such names are unbound and are looked up at the point of the template instantiation (13.8.5.1) in both the context of the template definition and the context of the point of instantiation (13.8.5.2). —end note]

2. [Example 1:

   template<class T> struct X : B<T> {
       typename T::A* pa;
       void f(B<T>::* pb) {
           static int i = B<T>::i;
           pb->j++;
       }
   };

   The base class name B<T>, the type name T::A, the names B<T>::i and pb->j explicitly depend on the template-parameter. —end example]
In the definition of a class or class template, the scope of a dependent base class (13.8.3.2) is not examined during unqualified name lookup either at the point of definition of the class template or member or during an instantiation of the class template or member.

[Example 2:

typedef double A;
template<class T> class B {
    typedef int A;
};
template<class T> struct X : B<T> {
    A a; // a has type double
};

The type name A in the definition of X<T> binds to the typedef name defined in the global namespace scope, not to the typedef name defined in the base class B<T>. — end example]

[Example 3:

struct A {
    struct B { /* ... */ };  
    int a;  
    int Y;
};

int a;

template<class T> struct Y : T {
    struct B { /* ... */ };  
    B b; // The B defined in Y  
    void f(int i) { a = i; } // ::a  
    Y* p; // Y<T>
};

Y<A> ya;

The members A::B, A::a, and A::Y of the template argument A do not affect the binding of names in Y<A>. — end example]

13.8.3.2 Dependent types [temp.dep.type]

1 A name refers to the current instantiation if it is

(1.1) — in the definition of a class template, a nested class of a class template, a member of a class template, or a member of a nested class of a class template, the injected-class-name (11.1) of the class template or nested class,

(1.2) — in the definition of a primary class template or a member of a primary class template, the name of the class template followed by the template argument list of the primary template (as described below) enclosed in <> (or an equivalent template alias specialization),

(1.3) — in the definition of a nested class of a class template, the name of the nested class referenced as a member of the current instantiation, or

(1.4) — in the definition of a partial specialization or a member of a partial specialization, the name of the class template followed by the template argument list of the partial specialization enclosed in <> (or an equivalent template alias specialization). If the n\text{th} template parameter is a template parameter pack, the n\text{th} template argument is a pack expansion (13.7.4) whose pattern is the name of the template parameter pack.

2 The template argument list of a primary template is a template argument list in which the n\text{th} template argument has the value of the n\text{th} template parameter of the class template. If the n\text{th} template parameter is a template parameter pack (13.7.4), the n\text{th} template argument is a pack expansion (13.7.4) whose pattern is the name of the template parameter pack.

3 A template argument that is equivalent to a template parameter can be used in place of that template parameter in a reference to the current instantiation. For a template type-parameter, a template argument is equivalent to a template parameter if it denotes the same type. For a non-type template parameter, a template argument is equivalent to a template parameter if it is an identifier that names a variable that is equivalent to the template parameter. A variable is equivalent to a template parameter if
— it has the same type as the template parameter (ignoring cv-qualification) and
— its initializer consists of a single identifier that names the template parameter or, recursively, such a
variable.

[Note 1: Using a parenthesized variable name breaks the equivalence. — end note]

[Example 1:

```cpp
template <class T> class A {
    A* p1; // A is the current instantiation
    A<T>* p2; // A is the current instantiation
    A<T*> p3; // A<T*> is not the current instantiation
    ::A<T>* p4; // ::A<T> is the current instantiation

    class B {
        B* p1; // B is the current instantiation
        A<T>::B* p2; // A<T>::B is the current instantiation
        typename A<T>::B* p3; // A<T>::B is not the current instantiation
    }
};

template <class T> class A<T*> {
    A<T*>* p1; // A<T*> is the current instantiation
    A<T>* p2; // A<T> is not the current instantiation
};

template <class T1, class T2, int I> struct B {
    B<T1, T2, I>* b1; // refers to the current instantiation
    B<T2, T1, I>* b2; // not the current instantiation
    typedef T1 my_T1;
    static const int my_I = I;
    static const int my_I2 = I+0;
    static const int my_I3 = my_I;
    static const long my_I4 = I;
    static const int my_I5 = (I);
    B<my_T1, T2, my_I>* b3; // refers to the current instantiation
    B<my_T1, T2, my_I2>* b4; // not the current instantiation
    B<my_T1, T2, my_I3>* b5; // refers to the current instantiation
    B<my_T1, T2, my_I4>* b6; // not the current instantiation
    B<my_T1, T2, my_I5>* b7; // not the current instantiation
};
```
—end example]

4 A dependent base class is a base class that is a dependent type and is not the current instantiation.

[Note 2: A base class can be the current instantiation in the case of a nested class naming an enclosing class as a base.

[Example 2:

```cpp
template<class T> struct A {
    typedef int M;
    struct B {
        typedef void M;
        struct C;
    }
};

template<class T> struct A<T>::B::C : A<T> {
    M m; // OK, A<T>::M
};
—end example]

—end note]

5 A name is a member of the current instantiation if it is
— An unqualified name that, when looked up, refers to at least one member of a class that is the current
instantiation or a non-dependent base class thereof.

§ 13.8.3.2
A name is a **member of an unknown specialization** if it is

- **(6.1)** A qualified-id in which the nested-name-specifier names a dependent type that is not the current instantiation.

- **(6.2)** A qualified-id in which the nested-name-specifier refers to the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the qualified-id does not find any member of a class that is the current instantiation or a non-dependent base class thereof.

- **(6.3)** An *id-expression* denoting the member in a class member access expression (7.6.1.5) in which either
  
  - the type of the object expression is the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the *id-expression* does not find a member of a class that is the current instantiation or a non-dependent base class thereof; or
  
  - the type of the object expression is not the current instantiation and the object expression is type-dependent.

If a qualified-id in which the nested-name-specifier refers to the current instantiation is not a member of the current instantiation or a member of an unknown specialization, the program is ill-formed even if the template containing the qualified-id is not instantiated; no diagnostic required. Similarly, if the *id-expression* in a class member access expression for which the type of the object expression is the current instantiation does not refer to a member of the current instantiation or a member of an unknown specialization, the program is ill-formed even if the template containing the member access expression is not instantiated; no diagnostic required.

*Example 4:*

```cpp
template<class T> class A {
  typedef int type;

  void f() {
    A<T>::type i; // OK: refers to a member of the current instantiation
    typename A<T>::other j; // error: neither a member of the current instantiation nor a member of an unknown specialization
  }
};
```
If, for a given set of template arguments, a specialization of a template is instantiated that refers to a member of the current instantiation with a qualified-id or class member access expression, the name in the qualified-id or class member access expression is looked up in the template instantiation context. If the result of this lookup differs from the result of name lookup in the template definition context, name lookup is ambiguous.

Example 5:

```cpp
struct A {
    int m;
};
struct B {
    int m;
};
template<typename T>
struct C : A, T {
    int f() { return this->m; } // finds A::m in the template definition context
    int g() { return m; }      // finds A::m in the template definition context
};
template int C<B>::f();       // error: finds both A::m and B::m
template int C<B>::g();       // OK: transformation to class member access syntax
```

A type is dependent if it is

(9.1) a template parameter,
(9.2) a member of an unknown specialization,
(9.3) a nested class or enumeration that is a dependent member of the current instantiation,
(9.4) a cv-qualified type where the cv-unqualified type is dependent,
(9.5) a compound type constructed from any dependent type,
(9.6) an array type whose element type is dependent or whose bound (if any) is value-dependent,
(9.7) a function type whose exception specification is value-dependent,
(9.8) denoted by a simple-template-id in which either the template name is a template parameter or any of the template arguments is a dependent type or an expression that is type-dependent or value-dependent or is a pack expansion,\(^\text{141}\) or
(9.9) denoted by decltype(expression), where expression is type-dependent (13.8.3.3).

Note 6: Because typedefs do not introduce new types, but instead simply refer to other types, a name that refers to a typedef that is a member of the current instantiation is dependent only if the type referred to is dependent. — end note

### Type-dependent expressions

Except as described below, an expression is type-dependent if any subexpression is type-dependent.

**this** is type-dependent if the class type of the enclosing member function is dependent (13.8.3.2).

An id-expression is type-dependent if it is not a concept-id and it contains

(3.1) an identifier associated by name lookup with one or more declarations declared with a dependent type,
(3.2) an identifier associated by name lookup with a non-type template-parameter declared with a type that contains a placeholder type (9.2.9.6),
(3.3) an identifier associated by name lookup with a variable declared with a type that contains a placeholder type (9.2.9.6) where the initializer is type-dependent,
(3.4) an identifier associated by name lookup with one or more declarations of member functions of the current instantiation declared with a return type that contains a placeholder type,

\(^{141}\) This includes an injected-class-name (11.1) of a class template used without a template-argument-list.
— an identifier associated by name lookup with a structured binding declaration (9.6) whose brace-or-
equal-initializer is type-dependent,

— the identifier __func__ (9.5.1), where any enclosing function is a template, a member of a class template,
or a generic lambda,

— a template-id that is dependent,

— a conversion-function-id that specifies a dependent type, or

— a nested-name-specifier or a qualified-id that names a member of an unknown specialization;

or if it names a dependent member of the current instantiation that is a static data member of type “array of
unknown bound of T” for some T (13.7.2.5). Expressions of the following forms are type-dependent only if
the type specified by the type-id, simple-type-specifier or new-type-id is dependent, even if any subexpression
is type-dependent:

\[
\begin{align*}
\text{simple-type-specifier} & \ (\text{expression-list}_{\text{opt}}) \\
\text{::opt} & \ \text{new-placement}_{\text{opt}} \ 	ext{new-type-id} \ \text{new-initializer}_{\text{opt}} \\
\text{::opt} & \ \text{new-placement}_{\text{opt}} \ (\text{type-id}) \ \text{new-initializer}_{\text{opt}} \\
\text{dynamic_cast} & \ <\text{type-id}> \ (\text{expression}) \\
\text{static_cast} & \ <\text{type-id}> \ (\text{expression}) \\
\text{const_cast} & \ <\text{type-id}> \ (\text{expression}) \\
\text{reinterpret_cast} & \ <\text{type-id}> \ (\text{type-id}) \ \text{cast-expression}
\end{align*}
\]

Expressions of the following forms are never type-dependent (because the type of the expression cannot be
dependent):

\[
\begin{align*}
\text{literal} \\
\text{sizeof} & \ \text{unary-expression} \\
\text{sizeof} & \ (\text{type-id}) \\
\text{sizeof} & \ldots \ (\text{identifier}) \\
\text{alignof} & \ (\text{type-id}) \\
\text{typeid} & \ (\text{expression}) \\
\text{typeid} & \ (\text{type-id}) \ \text{::opt} \ \text{delete} \ \text{cast-expression} \\
\text{::opt} & \ \text{delete} \ [ \ ] \ \text{cast-expression} \\
\text{throw} & \ \text{assignment-expression}_{\text{opt}} \\
\text{noexcept} & \ (\text{expression})
\end{align*}
\]

[Note 1: For the standard library macro offsetof, see 17.2. — end note]

A class member access expression (7.6.1.5) is type-dependent if the expression refers to a member of the
current instantiation and the type of the referenced member is dependent, or the class member access
expression refers to a member of an unknown specialization.

[Note 2: In an expression of the form x.y or xp->y the type of the expression is usually the type of the member y
of the class of x (or the class pointed to by xp). However, if x or xp refers to a dependent type that is not the
current instantiation, the type of y is always dependent. If x or xp refers to a non-dependent type or refers to the
current instantiation, the type of y is the type of the class member access expression. — end note]

A braced-init-list is type-dependent if any element is type-dependent or is a pack expansion.

A fold-expression is type-dependent.

13.8.3.4 Value-dependent expressions [temp.dep.constexpr]

Except as described below, an expression used in a context where a constant expression is required is
value-dependent if any subexpression is value-dependent.

An id-expression is value-dependent if:

(2.1) — it is a concept-id and any of its arguments are dependent,

(2.2) — it is type-dependent,

(2.3) — it is the name of a non-type template parameter,

(2.4) — it names a static data member that is a dependent member of the current instantiation and is not
initialized in a member-declarator,

(2.5) — it names a static member function that is a dependent member of the current instantiation, or
— it names a potentially-constant variable (7.7) that is initialized with an expression that is value-dependent.

Expressions of the following form are value-dependent if the unary-expression or expression is type-dependent or the type-id is dependent:

- sizeof unary-expression
- sizeof ( type-id )
- typeid ( expression )
- typeid ( type-id )
- alignof ( type-id )
- noexcept ( expression )

[Note 1: For the standard library macro offsetof, see 17.2. — end note]

3 Expressions of the following form are value-dependent if either the type-id or simple-type-specifier is dependent or the expression or cast-expression is value-dependent:

- simple-type-specifier ( expression-list_opt )
- static_cast < type-id > ( expression )
- const_cast < type-id > ( expression )
- reinterpret_cast < type-id > ( expression )
- ( type-id ) cast-expression

4 Expressions of the following form are value-dependent:

- sizeof ... ( identifier )
- fold-expression

5 An expression of the form &qualified-id where the qualified-id names a dependent member of the current instantiation is value-dependent. An expression of the form &cast-expression is also value-dependent if evaluating cast-expression as a core constant expression (7.7) succeeds and the result of the evaluation refers to a templated entity that is an object with static or thread storage duration or a member function.

13.8.3.5 Dependent template arguments

1 A type template-argument is dependent if the type it specifies is dependent.

2 A non-type template-argument is dependent if its type is dependent or the constant expression it specifies is value-dependent.

3 Furthermore, a non-type template-argument is dependent if the corresponding non-type template-parameter is of reference or pointer type and the template-argument designates or points to a member of the current instantiation or a member of a dependent type.

4 A template template-argument is dependent if it names a template-parameter or is a qualified-id that refers to a member of an unknown specialization.

13.8.4 Non-dependent names

1 Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used.

[Example 1:]

```c
void g(double);
void h();

template<class T> class Z {
    public:
    void f() {
        g(1); // calls g(double)
        h++; // ill-formed: cannot increment function; this could be diagnosed
              // either here or at the point of instantiation
    }
};

void g(int); // not in scope at the point of the template definition, not considered for the call g(1)
```

— end example]
13.8.5 Dependent name resolution

13.8.5.1 Point of instantiation

1 For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

2 If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

3 For a noexcept-specifier of a function template specialization or specialization of a member function of a class template, if the noexcept-specifier is implicitly instantiated because it is needed by another template specialization and the context that requires it depends on a template parameter, the point of instantiation of the noexcept-specifier is the point of instantiation of the specialization that requires it. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that requires the noexcept-specifier.

4 For a class template specialization, a class member template specialization, or a specialization for a class member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization, if the context from which the specialization is referenced depends on a template parameter, and if the specialization is not instantiated previous to the instantiation of the enclosing template, the point of instantiation is immediately before the point of instantiation of the enclosing template. Otherwise, the point of instantiation for such a specialization immediately precedes the namespace scope declaration or definition that refers to the specialization.

5 If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.

6 An explicit instantiation definition is an instantiation point for the specialization or specializations specified by the explicit instantiation.

7 A specialization for a function template, a member function template, or of a member function or static data member of a class template may have multiple points of instantiations within a translation unit, and in addition to the points of instantiation described above,

(7.1) for any such specialization that has a point of instantiation within the declaration-seq of the translation-unit, prior to the private-module-fragment (if any), the point after the declaration-seq of the translation-unit is also considered a point of instantiation, and

(7.2) for any such specialization that has a point of instantiation within the private-module-fragment, the end of the translation unit is also considered a point of instantiation.

A specialization for a class template has at most one point of instantiation within a translation unit. A specialization for any template may have points of instantiation in multiple translation units. If two different points of instantiation give a template specialization different meanings according to the one-definition rule (6.3), the program is ill-formed, no diagnostic required.

13.8.5.2 Candidate functions

1 For a function call where the postfix-expression is a dependent name, the candidate functions are found using the usual lookup rules from the template definition context (6.5.2, 6.5.3).

[Note 1: For the part of the lookup using associated namespaces (6.5.3), function declarations found in the template instantiation context are found by this lookup, as described in 6.5.3. — end note]

If the call would be ill-formed or would find a better match had the lookup within the associated namespaces considered all the function declarations with external linkage introduced in those namespaces in all translation units, not just considering those declarations found in the template definition and template instantiation contexts, then the program has undefined behavior.

[Example 1:}
Source file "X.h":
   namespace Q {
      struct X { };
   }

Source file "G.h":
   namespace Q {
      void g_impl(X, X);
   }

Module interface unit of M1:
   module;
   #include "X.h"
   #include "G.h"
   export module M1;
   export template<typename T>
   void g(T t) {
      g_impl(t, Q::X{ }); // ADL in definition context finds Q::g_impl, g_impl not discarded
   }

Module interface unit of M2:
   module;
   #include "X.h"
   export module M2;
   import M1;
   void h(Q::X x) {
      g(x); // OK
   }

   — end example]

Module interface unit of Std:
   export module Std;
   export template<typename Iter>
   void indirect_swap(Iter lhs, Iter rhs)
   {
      swap(*lhs, *rhs); // swap not found by unqualified lookup, can be found only via ADL
   }

Module interface unit of M:
   export module M;
   import Std;

      struct S { /* ... */ }; // #1
   void swap(S&, S&);         // #1
   void f(S* p, S* q)
   {
      indirect_swap(p, q); // finds #1 via ADL in instantiation context
   }

   — end example]

Module interface unit of F:
   export module F;
   export template<typename T>
   void f(T t) {
      t + t;
   }

§ 13.8.5.2
Module interface unit of $M$:

```c
module;
#include "X.h"
export module M;
import F;
void g(X x) {
    f(x);      // OK: instantiates $t$ from $F$,
                // operator+ is visible in instantiation context
}
```
—end example

Module interface unit of $A$:

```c
export module A;
export template<typename T>
void f(T t) {
    cat(t, t);        // #1
    dog(t, t);        // #2
}
```

Module interface unit of $B$:

```c
export module B;
import A;
export template<typename T, typename U>
void g(T t, U u) {
    f(t);
}
```

Source file "foo.h", not an importable header:

```c
struct foo {
    friend int cat(foo, foo);
};
int dog(foo, foo);
```

Module interface unit of $C1$:

```
module;
#include "foo.h"    // dog not referenced, discarded
export module C1;
import B;
export template<typename T>
void h(T t) {
    g(foo{ }, t);
}
```

Translation unit:

```c
import C1;
void i() {
    h(0);            // error: dog not found at #2
}
```

Importable header "bar.h":

```c
struct bar {
    friend int cat(bar, bar);
};
int dog(bar, bar);
```

Module interface unit of $C2$:

```
module;
#include "bar.h"    // imports header unit "bar.h"
export module C2;
import B;
```
export template<typename T>
void j(T t) {
    g(bar{ }, t);
}

Translation unit:
import C2;
void k() {
    j(0);
    // OK, dog found in instantiation context:
    // visible at end of module interface unit of C2
}
—end example]

13.8.6 Friend names declared within a class template [temp.inject]

Friend classes or functions can be declared within a class template. When a template is instantiated, the
names of its friends are treated as if the specialization had been explicitly declared at its point of instantiation.

As with non-template classes, the names of namespace-scope friend functions of a class template specialization
are not visible during an ordinary lookup unless explicitly declared at namespace scope (11.9.4). Such names
may be found under the rules for associated classes (6.5.3).

Example 1:

    template<typename T> struct number {
        number(int);
        friend number gcd(number x, number y) { return 0; };
    };

    void g() {
        number<double> a(3), b(4);
        a = gcd(a,b); // finds gcd because number<double> is an associated class,
    // making gcd visible in its namespace (global scope)
        b = gcd(3,4); // error: gcd is not visible
    }
—end example]

13.9 Template instantiation and specialization [temp.spec]

13.9.1 General [temp.spec.general]

The act of instantiating a function, a variable, a class, a member of a class template, or a member template
is referred to as template instantiation.

A function instantiated from a function template is called an instantiated function. A class instantiated from
a class template is called an instantiated class. A member function, a member class, a member enumeration,
or a static data member of a class template instantiated from the member definition of the class template is
called, respectively, an instantiated member function, member class, member enumeration, or static data
member. A member function instantiated from a member function template is called an instantiated member
function. A member class instantiated from a member class template is called an instantiated member class.
A variable instantiated from a variable template is called an instantiated variable. A static data member
instantiated from a static data member template is called an instantiated static data member.

An explicit specialization may be declared for a function template, a variable template, a class template, a
member of a class template, or a member template. An explicit specialization declaration is introduced by
template<>. In an explicit specialization declaration for a variable template, a class template, a member of
a class template or a class member template, the name of the variable or class that is explicitly specialized
shall be a simple-template-id. In the explicit specialization declaration for a function template or a member
function template, the name of the function or member function explicitly specialized may be a template-id.

Example 1:

    template<class T = int> struct A {
        static int x;
    };

142) Friend declarations do not introduce new names into any scope, either when the template is declared or when it is
instantiated.

§ 13.9.1

418
template<class U> void g(U) { }

template<> struct A<double> { };  // specialize for T == double
template<> struct A<> { };        // specialize for T == int
// U is deduced from the parameter type

template<> void g(char) { }        // specialize for U == char
// U is deduced from the parameter type

template<class T = int> struct B {
  static int x;
};
template<> int B<>::x = 1;          // specialize for T == int
— end example

4 An instantiated template specialization can be either implicitly instantiated (13.9.2) for a given argument list or be explicitly instantiated (13.9.3). A specialization is a class, variable, function, or class member that is either instantiated (13.9.2) from a templated entity or is an explicit specialization (13.9.4) of a templated entity.

5 For a given template and a given set of template-arguments,

(5.1) — an explicit instantiation definition shall appear at most once in a program,
(5.2) — an explicit specialization shall be defined at most once in a program, as specified in 6.3, and
(5.3) — both an explicit instantiation and a declaration of an explicit specialization shall not appear in a program unless the explicit instantiation follows a declaration of the explicit specialization.

An implementation is not required to diagnose a violation of this rule.

6 The usual access checking rules do not apply to names in a declaration of an explicit instantiation or explicit specialization, with the exception of names appearing in a function body, default argument, base-clause, member-specification, enumerator-list, or static data member or variable template initializer.

[Note 1: In particular, the template arguments and names used in the function declarator (including parameter types, return types and exception specifications) can be private types or objects that would normally not be accessible. — end note]

7 Each class template specialization instantiated from a template has its own copy of any static members.

[Example 2:]
  template<class T> class X {
    static T s;
  };  
template<class T> T X<T>::s = 0;
X<int> aa;
X<char*> bb;
X<int> has a static member s of type int and X<char*> has a static member s of type char*. — end example]

8 If a function declaration acquired its function type through a dependent type (13.8.3.2) without using the syntax of a function declarator, the program is ill-formed.

[Example 3:]
  template<class T> struct A {
    static T t;
  };  
typedef int function();
A<function> a;       // error: would declare A<function>::t as a static member function
— end example]

13.9.2 Implicit instantiation [temp.inst]

A template specialization E is a declared specialization if there is a reachable explicit instantiation definition (13.9.3) or explicit specialization declaration (13.9.4) for E, or if there is a reachable explicit instantiation declaration for E and E is not

(1.1) — an inline function,
— declared with a type deduced from its initializer or return value (9.2.9.6),
— a potentially-constant variable (7.7), or
— a specialization of a templated class.

[Note 1: An implicit instantiation in an importing translation unit cannot use names with internal linkage from an imported translation unit (6.6). — end note]

2 Unless a class template specialization is a declared specialization, the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program.

[Note 2: In particular, if the semantics of an expression depend on the member or base class lists of a class template specialization, the class template specialization is implicitly generated. For instance, deleting a pointer to class type depends on whether or not the class declares a destructor, and a conversion between pointers to class type depends on the inheritance relationship between the two classes involved. — end note]

[Example 1:]

```
template<class T> class B { /* ... */ };  
template<class T> class D : public B<T> { /* ... */ };  

void f(void*);  
void f(B<int>*);

void g(D<int>* p, D<char>* pp, D<double>* ppp) {  
f(p); // instantiation of D<int> required: call f(B<int>*)  
B<char>* q = pp; // instantiation of D<char> required: convert D<char>* to B<char>*  
delete ppp; // instantiation of D<double> required
}
```

— end example]

If a class template has been declared, but not defined, at the point of instantiation (13.8.5.1), the instantiation yields an incomplete class type (6.8).

[Example 2:]

```
template<class T> class X;  
X<char> ch; // error: incomplete type

template<> void C<int>::g() { } // error: redefinition of C<int>::g
```

— end example]

[Note 3: Within a template declaration, a local class (11.6) or enumeration and the members of a local class are never considered to be entities that can be separately instantiated (this includes their default arguments, noexcept-specifiers, and non-static data member initializers, if any, but not their type-constraints or requires-clauses). As a result, the dependent names are looked up, the semantic constraints are checked, and any templates used are instantiated as part of the instantiation of the entity within which the local class or enumeration is declared. — end note]

3 The implicit instantiation of a class template specialization causes

— the implicit instantiation of the declarations, but not of the definitions, of the non-deleted class member functions, member classes, scoped member enumerations, static data members, member templates, and friends; and

— the implicit instantiation of the definitions of deleted member functions, unscoped member enumerations, and member anonymous unions.

The implicit instantiation of a class template specialization does not cause the implicit instantiation of default arguments or noexcept-specifiers of the class member functions.

[Example 3:]

```
template<class T>  
struct C {  
  void f() { T x; }  
  void g() = delete;  
};

C<void> c; // OK, definition of C<void>::f is not instantiated at this point

template<> void C<int>::g() { } // error: redefinition of C<int>::g
```

— end example]
However, for the purpose of determining whether an instantiated redeclaration is valid according to 6.3 and 11.4, a declaration that corresponds to a definition in the template is considered to be a definition.

[Example 4:]

```cpp
template<class T, class U>
struct Outer {
    template<class X, class Y> struct Inner;
    template<class Y> struct Inner<T, Y>;  // #1a
    template<class Y> struct Inner<T, Y> { };  // #1b; OK: valid redeclaration of #1a
    template<class Y> struct Inner<U, Y> { };  // #2
};

Outer<int, int> outer;  // error at #2

Outer<int, int>::Inner<int, Y> is redeclared at #1b. (It is not defined but noted as being associated with a definition in Outer<T, U>.) #2 is also a redeclaration of #1a. It is noted as associated with a definition, so it is an invalid redeclaration of the same partial specialization.

template<typename T> struct Friendly {
    template<typename U> friend int f(U) { return sizeof(T); }
};

Friendly<char> fc;
Friendly<float> ff;  // error: produces second definition of f(U)
```

—end example

4 Unless a member of a class template or a member template is a declared specialization, the specialization of the member is implicitly instantiated when the specialization is referenced in a context that requires the member definition to exist or if the existence of the definition of the member affects the semantics of the program; in particular, the initialization (and any associated side effects) of a static data member does not occur unless the static data member is itself used in a way that requires the definition of the static data member to exist.

5 Unless a function template specialization is a declared specialization, the function template specialization is implicitly instantiated when the specialization is referenced in a context that requires a function definition to exist or if the existence of the definition affects the semantics of the program. A function whose declaration was instantiated from a friend function definition is implicitly instantiated when it is referenced in a context that requires a function definition to exist or if the existence of the definition affects the semantics of the program. Unless a call is to a function template explicit specialization or to a member function of an explicitly specialized class template, a default argument for a function template or a member function of a class template is implicitly instantiated when the function is called in a context that requires the value of the default argument.

[Note 1: An inline function that is the subject of an explicit instantiation declaration is not a declared specialization; the intent is that it still be implicitly instantiated when odr-used (6.3) so that the body can be considered for inlining, but that no out-of-line copy of it be generated in the translation unit. — end note]

6 [Example 5:]

```cpp
template<class T> struct Z {
    void f();
    void g();
};

void h() {
    Z<int> a;  // instantiation of class Z<int> required
    Z<char>* p;  // instantiation of class Z<char> not required
    Z<double>* q;  // instantiation of class Z<double> not required

    a.f();  // instantiation of Z<int>::f() required
    p->g();  // instantiation of class Z<char> required, and
              // instantiation of Z<char>::g() required
}
```

Nothing in this example requires class Z<double>, Z<int>::g(), or Z<char>::f() to be implicitly instantiated. —end example]
Unless a variable template specialization is a declared specialization, the variable template specialization is implicitly instantiated when it is referenced in a context that requires a variable definition to exist or if the existence of the definition affects the semantics of the program. A default template argument for a variable template is implicitly instantiated when the variable template is referenced in a context that requires the value of the default argument.

The existence of a definition of a variable or function is considered to affect the semantics of the program if the variable or function is needed for constant evaluation by an expression (7.7), even if constant evaluation of the expression is not required or if constant expression evaluation does not use the definition.

[Example 6:
```cpp
template<typename T> constexpr int f() { return T::value; }
template<bool B, typename T> void gdecltype(B ? f<T>() : 0));
template<bool B, typename T> void g(...);
template<bool B, typename T> void hdecltype(int{B ? f<T>() : 0})));
template<bool B, typename T> void h(...);
void x() {
g<false, int>(0); // OK, B ? f<T>() : 0 is not potentially constant evaluated
h<false, int>(0); // error, instantiates f<int> even though B evaluates to false and
// list-initialization of int from int cannot be narrowing
}
—end example]

If the function selected by overload resolution (12.4) can be determined without instantiating a class template definition, it is unspecified whether that instantiation actually takes place.

[Example 7:
```cpp
template <class T> struct S {
    operator int();
};
void f(int);
void f(S<int>&);
void f(S<float>);
void g(S<int>& sr) {
f(sr);  // instantiation of S<int> allowed but not required
// instantiation of S<float> allowed but not required
};
—end example]

If a function template or a member function template specialization is used in a way that involves overload resolution, a declaration of the specialization is implicitly instantiated (13.10.4).

An implementation shall not implicitly instantiate a function template, a variable template, a member template, a non-virtual member function, a member class, a static data member of a class template, or a substatement of a constexpr if statement (8.5.2), unless such instantiation is required.

[Note 5: The instantiation of a generic lambda does not require instantiation of substatements of a constexpr if statement (8.5.2), unless such instantiation is required. — end note]

It is unspecified whether or not an implementation implicitly instantiates a virtual member function of a class template if the virtual member function would not otherwise be instantiated. The use of a template specialization in a default argument shall not cause the template to be implicitly instantiated except that a class template may be instantiated where its complete type is needed to determine the correctness of the default argument. The use of a default argument in a function call causes specializations in the default argument to be implicitly instantiated.

Implicitly instantiated class, function, and variable template specializations are placed in the namespace where the template is defined. Implicitly instantiated specializations for members of a class template are placed in the namespace where the enclosing class template is defined. Implicitly instantiated member templates are placed in the namespace where the enclosing class or class template is defined.

[Example 8:
namespace N {
    template<class T> class List {
    public:
        T* get();
    }
}

template<class K, class V> class Map {
    public:
        N::List<V> lt;
        V get(K);
    }

void g(Map<const char*,int>& m) {
    int i = m.get("Nicholas");
}

A call of `lt.get()` from `Map<const char*,int>::get()` would place `List<int>::get()` in the namespace `N` rather than in the global namespace. —end example—

If a function template `f` is called in a way that requires a default argument to be used, the dependent names are looked up, the semantics constraints are checked, and the instantiation of any template used in the default argument is done as if the default argument had been an initializer used in a function template specialization with the same scope, the same template parameters and the same access as that of the function template `f` used at that point, except that the scope in which a closure type is declared (7.5.5.2) – and therefore its associated namespaces – remain as determined from the context of the definition for the default argument. This analysis is called default argument instantiation. The instantiated default argument is then used as the argument of `f`.

Each default argument is instantiated independently.

[Example 9]:
```
template<class T> void f(T x, T y = ydef(T()), T z = zdef(T()));

class A { }

A zdef(A);
```

void g(A a, A b, A c) {
    f(a, b, c);       // no default argument instantiation
    f(a, b);         // default argument `z = zdef(T())` instantiated
    f(a);            // error: `ydef` is not declared
}
—end example—

The noexcept-specifier of a function template specialization is not instantiated along with the function declaration; it is instantiated when needed (14.5). If such an noexcept-specifier is needed but has not yet been instantiated, the dependent names are looked up, the semantics constraints are checked, and the instantiation of any template used in the noexcept-specifier is done as if it were being done as part of instantiating the declaration of the specialization at that point.

[Note 6: 13.8.5.1 defines the point of instantiation of a template specialization. —end note]

There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations (Annex B), which could involve more than one template. The result of an infinite recursion in instantiation is undefined.

[Example 10]:
```
template<class T> class X {
    X<T*>* p;        // OK
    X<T*> a;         // implicit generation of `X<T>` requires
                     // the implicit instantiation of `X<T*>` which requires
                     // the implicit instantiation of `X<T**>` which . . .
};
—end example—

§ 13.9.2
The type-constraints and requires-clause of a template specialization or member function are not instantiated along with the specialization or function itself, even for a member function of a local class; substitution into the atomic constraints formed from them is instead performed as specified in 13.5.3 and 13.5.2.3 when determining whether the constraints are satisfied or as specified in 13.5.3 when comparing declarations.

[Note 7: The satisfaction of constraints is determined during template argument deduction (13.10.3) and overload resolution (12.4). — end note]

[Example 11:

```cpp
template<typename T> concept C = sizeof(T) > 2;
template<typename T> concept D = C<T> && sizeof(T) > 4;

template<typename T> struct S {
    S() requires C<T> { } // #1
    S() requires D<T> { } // #2
};

S<char> s1; // error: no matching constructor
S<char[8]> s2; // OK, calls #2
```

When S<char> is instantiated, both constructors are part of the specialization. Their constraints are not satisfied, and they suppress the implicit declaration of a default constructor for S<char> (11.4.5.2), so there is no viable constructor for s1. — end example]

[Example 12:

```cpp
template<typename T> struct S1 {
    template<typename U>
    requires false
    struct Inner1; // ill-formed, no diagnostic required
};
template<typename T> struct S2 {
    template<typename U>
    requires (sizeof(T[-(int)sizeof(T)]) > 1)
    struct Inner2; // ill-formed, no diagnostic required
};
```

The class S1<T>::Inner1 is ill-formed, no diagnostic required, because it has no valid specializations. S2 is ill-formed, no diagnostic required, since no substitution into the constraints of its Inner2 template would result in a valid expression. — end example]

### 13.9.3 Explicit instantiation [temp.explicit]

A class, function, variable, or member template specialization can be explicitly instantiated from its template. A member function, member class or static data member of a class template can be explicitly instantiated from the member definition associated with its class template.

The syntax for explicit instantiation is:

```cpp
explicit-instantiation:
    extern opt template declaration
```

There are two forms of explicit instantiation: an explicit instantiation definition and an explicit instantiation declaration. An explicit instantiation declaration begins with the `extern` keyword.

An explicit instantiation shall not use a `storage-class-specifier` (9.2.2) other than `thread_local`. An explicit instantiation of a function template, member function of a class template, or variable template shall not use the `inline`, `constexpr`, or `consteval` specifiers. No `attribute-specifier-seq` (9.12.1) shall appertain to an explicit instantiation.

If the explicit instantiation is for a class or member class, the `elaborated-type-specifier` in the declaration shall include a `simple-template-id`; otherwise, the declaration shall be a `simple-declaration` whose `init-declarator-list` comprises a single `init-declarator` that does not have an `initializer`. If the explicit instantiation is for a function or member function, the `unqualified-id` in the declarator shall be either a `template-id` or, where all template arguments can be deduced, a `template-name` or `operator-function-id`.

[Note 1: The declaration can declare a `qualified-id`, in which case the `unqualified-id` of the `qualified-id` must be a `template-id`. — end note]
If the explicit instantiation is for a member function, a member class or a static data member of a class template specialization, the name of the class template specialization in the qualified-id for the member name shall be a simple-template-id. If the explicit instantiation is for a variable template specialization, the unqualified-id in the declarator shall be a simple-template-id. An explicit instantiation shall appear in an enclosing namespace of its template. If the name declared in the explicit instantiation is an unqualified name, the explicit instantiation shall appear in the namespace where its template is declared or, if that namespace is inline (9.8.2), any namespace from its enclosing namespace set.

[Note 2: Regarding qualified names in declarators, see 9.3.4. — end note]

[Example 1:]

```c++
template<class T> class Array { void mf(); }
template class Array<char>; 
template void Array<int>::mf();

template<class T> void sort(Array<T>& v) { /* ... */ }
template void sort(Array<char>&);  // argument is deduced here
namespace N {
  template<class T> void f(T&) { }
} 
template void N::f<int>(int&);
— end example]
```

A declaration of a function template, a variable template, a member function or static data member of a class template, or a member function template of a class or class template shall precede an explicit instantiation of that entity. A definition of a class template, a member class of a class template, or a member class template of a class or class template shall precede an explicit instantiation of that entity unless the explicit instantiation is preceded by an explicit specialization of the entity with the same template arguments. If the declaration of the explicit instantiation names an implicitly-declared special member function (11.4.4), the program is ill-formed.

The declaration in an explicit-instantiation and the declaration produced by the corresponding substitution into the templated function, variable, or class are two declarations of the same entity.

[Note 3: These declarations are required to have matching types as specified in 6.6, except as specified in 14.5.]

[Example 2:]

```c++
template<typename T> T var = {};       
template float var<float>;   // OK, instantiated variable has type float 
template int var<int[16]>[1]; // OK, absence of major array bound is permitted
template int * var<int>;   // error: instantiated variable has type int

template<typename T> auto av = T();  
template int av<int>;  // OK, variable with type int can be redeclared with type auto

template<typename T> auto f () {}      
template void f<int>();   // error: function with deduced return type 
// redeclared with non-deduced return type (9.2.9.6)
— end example]
— end note]
```

Despite its syntactic form, the declaration in an explicit-instantiation for a variable is not itself a definition and does not conflict with the definition instantiated by an explicit instantiation definition for that variable.

For a given set of template arguments, if an explicit instantiation of a template appears after a declaration of an explicit specialization for that template, the explicit instantiation has no effect. Otherwise, for an explicit instantiation definition, the definition of a function template, a variable template, a member function template, or a member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

An explicit instantiation of a class, function template, or variable template specialization is placed in the namespace in which the template is defined. An explicit instantiation for a member of a class template is placed in the namespace where the enclosing class template is defined. An explicit instantiation for a member template is placed in the namespace where the enclosing class or class template is defined.
Example 3:

```cpp
namespace N {
    template<class T> class Y { void mf() { } ; }
}
```

```cpp
template class Y<int> ; // error: class template Y not visible in the global namespace
using N::Y;
```

```cpp
template class Y<int> ; // error: explicit instantiation outside of the namespace of the template
```

```cpp
template class N::Y<char*> ; // OK: explicit instantiation in namespace N
template void N::Y<double>::mf() ; // OK: explicit instantiation in namespace N
```

—end example

A trailing template-argument can be left unspecified in an explicit instantiation of a function template specialization or of a member function template specialization provided it can be deduced from the type of a function parameter (13.10.3).

Example 4:

```cpp
template<class T> class Array { /* ... */ };
```

```cpp
template<class T> void sort(Array<T>& v) { /* ... */ };
```

```cpp
// instantiate sort(Array<int>&) – template-argument deduced
```

```cpp
template void sort<>(Array<int>&);
```

—end example

Note 4: An explicit instantiation of a constrained template is required to satisfy that template’s associated constraints (13.5.3). The satisfaction of constraints is determined when forming the template name of an explicit instantiation in which all template arguments are specified (13.3), or, for explicit instantiations of function templates, during template argument deduction (13.10.3.7) when one or more trailing template arguments are left unspecified.

—end note

An explicit instantiation that names a class template specialization is also an explicit instantiation of the same kind (declaration or definition) of each of its members (not including members inherited from base classes and members that are templates) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, provided that the associated constraints, if any, of that member are satisfied by the template arguments of the explicit instantiation (13.5.3, 13.5.2), except as described below.

Note 5: In addition, it will typically be an explicit instantiation of certain implementation-dependent data about the class.

—end note

An explicit instantiation definition that names a class template specialization explicitly instantiates the class template specialization and is an explicit instantiation definition of only those members that have been defined at the point of instantiation.

An explicit instantiation of a prospective destructor (11.4.7) shall name the selected destructor of the class.

If an entity is the subject of both an explicit instantiation declaration and an explicit instantiation definition in the same translation unit, the definition shall follow the declaration. An entity that is the subject of an explicit instantiation declaration and that is also used in a way that would otherwise cause an implicit instantiation (13.9.2) in the translation unit shall be the subject of an explicit instantiation definition somewhere in the program; otherwise the program is ill-formed, no diagnostic required.

Note 6: This rule does apply to inline functions even though an explicit instantiation declaration of such an entity has no other normative effect. This is needed to ensure that if the address of an inline function is taken in a translation unit in which the implementation chose to suppress the out-of-line body, another translation unit will supply the body.

—end note

An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is not done.

Example 5:

```cpp
char* p = 0;
template<class T> T g(T x = &p) { return x ; }
```

```cpp
template int g<int>(int) ; // OK even though &p isn’t an int.
```
13.9.4 Explicit specialization

An explicit specialization of any of the following:

1. function template
2. class template
3. variable template
4. member function of a class template
5. static data member of a class template
6. member class of a class template
7. member enumeration of a class template
8. member class template of a class or class template
9. member function template of a class or class template

can be declared by a declaration introduced by `template<>;` that is:

```c++
explicit-specialization:
  template <> declaration
```

[Example 1:]

```c++
template<class T> class stream;

template<> class stream<char> { /* ... */ };

template<class T> class Array { /* ... */ };

template<class T> void sort(Array<T>& v) { /* ... */ }

// error:
X not a template

// OK:
X is a template
```

Given these declarations, `stream<char>` will be used as the definition of streams of `char`; other streams will be handled by class template specializations instantiated from the class template. Similarly, `sort<char*>` will be used as the sort function for arguments of type `Array<char*>`; other `Array` types will be sorted by functions generated from the template.

---

An explicit specialization shall not use a `storage-class-specifier` (9.2.2) other than `thread_local`.

An explicit specialization may be declared in any scope in which the corresponding primary template may be defined (9.8.2.3, 11.4, 13.7.3).

A declaration of a function template, class template, or variable template being explicitly specialized shall precede the declaration of the explicit specialization.

[Note 1: A declaration, but not a definition of the template is required. --- end note]

The definition of a class or class template shall precede the declaration of an explicit specialization for a member template of the class or class template.

[Example 2:]

```c++
template<> class X<int> { /* ... */ };

// error: X not a template

template<class T> class X;

template<> class X<char*> { /* ... */ };

// OK: X is a template
```

A member function, a member function template, a member class, a member enumeration, a member class template, a static data member, or a static data member template of a class template may be explicitly specialized for a class specialization that is implicitly instantiated; in this case, the definition of the class template shall precede the explicit specialization for the member of the class template. If such an explicit specialization for the member of a class template names an implicitly-declared special member function (11.4.4), the program is ill-formed.

A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the class template; instead, the member of the class template specialization shall itself be explicitly defined if its
A template, a member template or a member of a class template is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required. If the program does not provide a definition for an explicit specialization and either the specialization is used in a way that would cause an implicit instantiation to take place or the member is a virtual member function, the program is ill-formed, no diagnostic required. An implicit instantiation is never generated for an explicit specialization that is declared but not defined.

Example 4:

```cpp
class String { }
template<class T> class Array { /* ... */ };  
template<class T> void sort(Array<T>& v) { /* ... */ }
```
void f(Array<String>& v) {
    sort(v);        // use primary template sort(Array<T>& v), T is String
}

template<>
void sort<String>(Array<String>& v);   // error: specialization after use of primary template

// OK: sort<char*> not yet used

template<class T>
struct A {

    enum E : T;
    enum class S : T;
};

template<>
enum A<int>::E : int { eint };       // OK

template<>
enum class A<int>::S : int { sint };  // OK

template<class T>
enum A<T>::E : T { eT };             // error: A<char>::E was instantiated

template<class T>
enum class A<T>::S : T { sT };       // when A<char> was instantiated

// OK

template<>
enum class A<char>::S : char { schar }; // OK

—end example

8 The placement of explicit specialization declarations for function templates, class templates, variable templates, member functions of class templates, static data members of class templates, member classes of class templates, member enumerations of class templates, member class templates of class templates, member function templates of class templates, static data member templates of class templates, member functions of member templates of class templates, member functions of member templates of non-template classes, static data member templates of non-template classes, member function templates of member classes of class templates, etc., and the placement of partial specialization declarations of class templates, variable templates, member class templates of non-template classes, static data member templates of non-template classes, member class templates of class templates, etc., can affect whether a program is well-formed according to the relative positioning of the explicit specialization declarations and their points of instantiation in the translation unit as specified above and below. When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation.

9 A template explicit specialization is in the scope of the namespace in which the template was defined.

[Example 5:

namespace N {

    template<class T>
    class X { /* ... */ };  

    template<class T>
    class Y { /* ... */ };  

    template<>
    class X<int> { /* ... */ };    // OK: specialization in same namespace

    template<class T>
    class Y<double>;             // forward-declare intent to specialize for double
}

template<>
class N::Y<double> { /* ... */ }; // OK: specialization in enclosing namespace

template<>
class N::Y<short> { /* ... */ };  // OK: specialization in enclosing namespace

—end example]

10 A simple-template-id that names a class template explicit specialization that has been declared but not defined can be used exactly like the names of other incompletely-defined classes (6.8).

[Example 6:

    template<class T>
    class X;                  // X is a class template

    template<class T>
    class X<int>;             // OK: pointer to declared class X<int>

    X<int>* p;                  // error: object of incomplete class X<int>
    X<int> x;                   // OK: object of incomplete class X<int>

—end example]

11 A trailing template-argument can be left unspecified in the template-id naming an explicit function template specialization provided it can be deduced from the function argument type.

[Example 7:

    template<class T>
    class Array { /* ... */ };    // OK: pointer to declared class Array<T>

    template<class T>
    void sort(Array<T>& v);  // OK: object of incomplete class Array<T>

§ 13.9.4 429
template<> void sort(Array<int>&);

— end example]

[Note 2: An explicit specialization of a constrained template is required to satisfy that template’s associated constraints (13.5.3). The satisfaction of constraints is determined when forming the template name of an explicit specialization in which all template arguments are specified (13.3), or, for explicit specializations of function templates, during template argument deduction (13.10.3.7) when one or more trailing template arguments are left unspecified. — end note]

A function with the same name as a template and a type that exactly matches that of a template specialization is not an explicit specialization (13.7.7).

Whether an explicit specialization of a function or variable template is inline, constexpr, or an immediate function is determined by the explicit specialization and is independent of those properties of the template.

[Example 8:

```cpp
template<class T> void f(T) { /* ... */ }
template<class T> inline T g(T) { /* ... */ }

template<> inline void f<>(int) { /* ... */ }  // OK: inline
template<> int g<>(int) { /* ... */ }  // OK: not inline
— end example]

An explicit specialization of a static data member of a template or an explicit specialization of a static data member template is a definition if the declaration includes an initializer; otherwise, it is a declaration.

[Note 3: The definition of a static data member of a template for which default-initialization is desired can use functional cast notation (7.6.1.4):

```cpp
template<> X Q<int>::x;
// declaration

template<> X Q<int>::x ();
// error: declares a function

template<> X Q<int>::x = X();
// definition
— end note]

A member or a member template of a class template may be explicitly specialized for a given implicit instantiation of the class template, even if the member or member template is defined in the class template definition. An explicit specialization of a member or member template is specified using the syntax for explicit specialization.

[Example 9:

```cpp
template<class T> struct A {
    void f(T);
    template<class X1> void g1(T, X1);
    template<class X2> void g2(T, X2);
    void h(T) { }
};

// specialization
template<> void A<int>::f(int);

// out of class member template definition
template<class T> template<class X1> void A<T>::::g1(T, X1) { }

// member template specialization
template<> template<class X1> void A<int>::::g1(int, X1);

// member specialization even if defined in class definition
template<> void A<int>::::h(int) { }
```
A member or a member template may be nested within many enclosing class templates. In an explicit specialization for such a member, the member declaration shall be preceded by a `template<>` for each enclosing class template that is explicitly specialized.

[Example 10:]
```
template<class T1> class A {
    template<class T2> class B {
        void mf();
    };
    template<> template<> class A<int>::B<double>;
    template<> template<> void A<char>::B<char>::mf();
}
```

—end example—

In an explicit specialization declaration for a member of a class template or a member template that appears in namespace scope, the member template and some of its enclosing class templates may remain unspecialized, except that the declaration shall not explicitly specialize a class member template if its enclosing class templates are not explicitly specialized as well. In such an explicit specialization declaration, the keyword `template` followed by a `template-parameter-list` shall be provided instead of the `template<>` preceding the explicit specialization declaration of the member. The types of the `template-parameters` in the `template-parameter-list` shall be the same as those specified in the primary template definition.

[Example 11:]
```
template <class T1> class A { 
    template<class T2> class B { 
        template<class T3> void mf1(T3);
        void mf2();
    };
    template <> template <class X> 
    class A<int>::B { 
        template <class T> void mf1(T);
    };
    template <> template <> template<class T> 
    void A<int>::B<double>::mf1(T t) { } // error: B<double> is specialized but 
    // its enclosing class template A is not 
    template <class Y> template <> 
    void A<Y>::B<double>::mf2() { } // error: B<double> is specialized but 
    // its enclosing class template A is not 
}
```

—end example—

A specialization of a member function template, member class template, or static data member template of a non-specialized class template is itself a template.

An explicit specialization declaration shall not be a friend declaration.

Default function arguments shall not be specified in a declaration or a definition for one of the following explicit specializations:

(21.1) — the explicit specialization of a function template;

(21.2) — the explicit specialization of a member function template;

(21.3) — the explicit specialization of a member function of a class template where the class template specialization to which the member function specialization belongs is implicitly instantiated.

[Note 4: Default function arguments can be specified in the declaration or definition of a member function of a class template specialization that is explicitly specialized. —end note]

13.10 Function template specializations

13.10.1 General

A function instantiated from a function template is called a function template specialization; so is an explicit specialization of a function template. Template arguments can be explicitly specified when naming the function template specialization, deduced from the context (e.g., deduced from the function arguments in a call to the function template specialization, see 13.10.3), or obtained from default template arguments.
Each function template specialization instantiated from a template has its own copy of any static variable.

**Example 1:**

```c
template<class T> void f(T* p) {
    static T s;
};
void g(int a, char* b) {
    f(&a);    // calls f<int>(int*)
    f(&b);    // calls f<char*>(char**)  
}
```

Here `f<int>(int*)` has a static variable `s` of type `int` and `f<char*>(char**)` has a static variable `s` of type `char*`.

— end example]

### 13.10.2 Explicit template argument specification

Template arguments can be specified when referring to a function template specialization that is not a specialization of a constructor template by qualifying the function template name with the list of template-arguments in the same way as template-arguments are specified in uses of a class template specialization.

**Example 1:**

```c
template<class T> void sort(Array<T>& v);
void f(Array<dcomplex>& cv, Array<int>& ci) {
    sort<dcomplex>(cv);    // sort<Array<dcomplex>&)
    sort<int>(ci);         // sort<Array<int>&)  
}
```

and

```c
template<class U, class V> U convert(V v);
void g(double d) {
    int i = convert<int,double>(d);    // int convert(double)
    char c = convert<char,double>(d);  // char convert(double)  
}
```

— end example]

Template arguments shall not be specified when referring to a specialization of a constructor template (11.4.5, 6.5.4.2).

A template argument list may be specified when referring to a specialization of a function template

(3.1) — when a function is called,
(3.2) — when the address of a function is taken, when a function initializes a reference to function, or when a pointer to member function is formed,
(3.3) — in an explicit specialization,
(3.4) — in an explicit instantiation, or
(3.5) — in a friend declaration.

Trailing template arguments that can be deduced (13.10.3) or obtained from default template-arguments may be omitted from the list of explicit template-arguments. A trailing template parameter pack (13.7.4) not otherwise deduced will be deduced as an empty sequence of template arguments. If all of the template arguments can be deduced, they may all be omitted; in this case, the empty template argument list `<>` itself may also be omitted. In contexts where deduction is done and fails, or in contexts where deduction is not done, if a template argument list is specified and it, along with any default template arguments, identifies a single function template specialization, then the template-id is an lvalue for the function template specialization.

**Example 2:**

```c
template<class X, class Y> X f(Y);
template<class X, class Y, class ... Z> X g(Y);
void h() {
    int i = f<int>(5.6);    // Y deduced as double
    int j = f(5.6);         // error: X cannot be deduced
    f<void>(f<int, bool>); // Y for outer f deduced as int (*)(bool)
}
```

§ 13.10.2
f<void>(f<int>); // error: f<int> does not denote a single function template specialization
int x = g<int>(5.6); // Y deduced as double; Z deduced as an empty sequence
f<void>(g<int, bool>); // Y for outer f deduced as int (*)(bool),
                      // Z deduced as an empty sequence
}
—end example]

[Note 1: An empty template argument list can be used to indicate that a given use refers to a specialization of a function template even when a non-template function (9.3.4.6) is visible that would otherwise be used. For example:

```
template <class T> int f(T); // #1
int f(int); // #2
int k = f(1); // uses #2
int l = f<>(1); // uses #1
—end note]

Template arguments that are present shall be specified in the declaration order of their corresponding template-parameters. The template argument list shall not specify more template-arguments than there are corresponding template-parameters unless one of the template-parameters is a template parameter pack.

[Example 3:

template<class X, class Y, class Z> X f(Y,Z);
template<class ... Args> void f2();
void g() {
f<int, const char*, double>("aa",3.0);
f<int, const char*>("aa",3.0); // Z deduced as double
f<int>("aa", 3.0); // Y deduced as const char*; Z deduced as double
f("aa", 3.0); // error: X cannot be deduced
f2<char, short, int, long>(); // OK
}
—end example]

Implicit conversions (7.3) will be performed on a function argument to convert it to the type of the corresponding function parameter if the parameter type contains no template-parameters that participate in template argument deduction.

[Note 2: Template parameters do not participate in template argument deduction if they are explicitly specified. For example,

```
template<class T> void f(T);

class Complex {
    Complex(double);
};

void g() {
f<Complex>(1); // OK, means f<Complex>(Complex(1))
}
—end note]

[Note 3: Because the explicit template argument list follows the function template name, and because constructor templates (11.4.5) are named without using a function name (6.5.4.2), there is no way to provide an explicit template argument list for these function templates. — end note]

Template argument deduction can extend the sequence of template arguments corresponding to a template parameter pack, even when the sequence contains explicitly specified template arguments.

[Example 4:

template<class ... Types> void f(Types ... values);

void g() {
f<int*, float*>(0, 0, 0); // Types deduced as the sequence int*, float*, int
}
—end example]
13.10.3 Template argument deduction

13.10.3.1 General

When a function template specialization is referenced, all of the template arguments shall have values. The values can be explicitly specified or, in some cases, be deduced from the use or obtained from default template-arguments.

Example 1:

```c
void f(Array<dcomplex>& cv, Array<int>& ci) {
  sort(cv); // calls sort(Array<dcomplex>&)
  sort(ci); // calls sort(Array<int>&)
}
```

and

```c
void g(double d) {
  int i = convert<int>(d); // calls convert<int,double>(double)
  int c = convert<char>(d); // calls convert<char,double>(double)
}
```

—end example

2 When an explicit template argument list is specified, if the given template-id is not valid (13.3), type deduction fails. Otherwise, the specified template argument values are substituted for the corresponding template parameters as specified below.

3 After this substitution is performed, the function parameter type adjustments described in 9.3.4.6 are performed.

Example 2: A parameter type of “void (const int, int[5])” becomes “void(*)(int,int*)”. —end example

Note 1: A top-level qualifier in a function parameter declaration does not affect the function type but still affects the type of the function parameter variable within the function. —end note

Example 3:

```c
template <class T> void f(T t);
template <class X> void g(const X x);
template <class Z> void h(Z, Z*);
```

```c
int main() {
  // #1: function type is f(int), t is non const
  f<int>(1);

  // #2: function type is f(int), t is const
  f<const int>(1);

  // #3: function type is g(int), x is const
  g<int>(1);

  // #4: function type is g(int), x is const
  g<const int>(1);

  // #5: function type is h<int, const int*>(
  h<int, const int*>(1,0);
}
```

—end example

Note 2: f<int>(1) and f<const int>(1) call distinct functions even though both of the functions called have the same function type. —end note

4 The resulting substituted and adjusted function type is used as the type of the function template for template argument deduction. If a template argument has not been deduced and its corresponding template parameter has a default argument, the template argument is determined by substituting the template arguments determined for preceding template parameters into the default argument. If the substitution results in an invalid type, as described above, type deduction fails.

Example 4:

```c
template <class T, class U = double>
void f(T t = 0, U u = 0);
```
When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in the template parameter list of the template and the function type are replaced with the corresponding deduced or default argument values. If the substitution results in an invalid type, as described above, type deduction fails. If the function template has associated constraints (13.5.3), those constraints are checked for satisfaction. If the constraints are not satisfied, type deduction fails.

At certain points in the template argument deduction process it is necessary to take a function type that makes use of template parameters and replace those template parameters with the corresponding template arguments. This is done at the beginning of template argument deduction when any explicitly specified template arguments are substituted into the function type, and again at the end of template argument deduction when any template arguments that were deduced or obtained from default arguments are substituted.

The substitution occurs in all types and expressions that are used in the function type and in template parameter declarations. The expressions include not only constant expressions such as those that appear in array bounds or as nontype template arguments but also general expressions (i.e., non-constant expressions) inside sizeof, decltype, and other contexts that allow non-constant expressions. The substitution proceeds in lexical order and stops when a condition that causes deduction to fail is encountered. If substitution into different declarations of the same function template would cause template instantiations to occur in a different order or not at all, the program is ill-formed; no diagnostic required.

If a substitution results in an invalid type or expression, type deduction fails. An invalid type or expression is one that would be ill-formed, with a diagnostic required, if written using the substituted arguments. If no diagnostic is required, the program is still ill-formed. Access checking is done as part of the substitution process. Only invalid types and expressions in the immediate context of the function type, its template parameter types, and its explicit-specifier can result in a deduction failure.

A lambda-expression appearing in a function type or a template parameter is not considered part of the immediate context for the purposes of template argument deduction.
[Note 6: The intent is to avoid requiring implementations to deal with substitution failure involving arbitrary statements.]

[Example 6:]

```cpp
template <class T>
    auto f(T) -> decltype([]() { T::invalid; })();
void f(...);
    // error: invalid expression not part of the immediate context

template <class T, std::size_t = sizeof([]() { T::invalid; })>
    void g(T);
void g(...);
    // error: invalid expression not part of the immediate context

template <class T>
    auto h(T) -> decltype([]() { x = T::invalid; })();
void h(...);
    // error: invalid expression not part of the immediate context

template <class T>
    auto i(T) -> decltype([]() -> typename T::invalid { });
void i(...);
    // error: invalid expression not part of the immediate context

template <class T>
    auto j(T t) -> decltype([](auto x) -> decltype(x.invalid) { } (t));    // #1
void j(...);                                                             // #2
j(0);                                                                  // deduction fails on #1, calls #2
```

— end example]

— end note]

[Example 7:]

```cpp
struct X { };
struct Y {
    Y(X){}
};

template <class T> auto f(T t1, T t2) -> decltype(t1 + t2);       // #1
X f(Y, Y);                                                          // #2
X x1, x2;
X x3 = f(x1, x2);                                                   // deduction fails on #1 (cannot add X+X), calls #2
```

— end example]

[Note 7: Type deduction can fail for the following reasons:

(11.1)   — Attempting to instantiate a pack expansion containing multiple packs of differing lengths.

(11.2)   — Attempting to create an array with an element type that is `void`, a function type, or a reference type, or attempting to create an array with a size that is zero or negative.

[Example 8:]

```cpp
    template <class T> int f(T[5]);
    int I = f<int>(0);
    int j = f<void>(0);                                        // invalid array
```

— end example]

(11.3)   — Attempting to use a type that is not a class or enumeration type in a qualified name.

[Example 9:]

```cpp
    template <class T> int f(typename T::*);    
    int i = f<int>(0);
```

— end example]

(11.4)   — Attempting to use a type in a `nested-name-specifier of a qualified-id` when that type does not contain the specified member, or

§ 13.10.3.1 436
— the specified member is not a type where a type is required, or
— the specified member is not a template where a template is required, or
— the specified member is not a non-type where a non-type is required.

[Example 10:]

\[
\begin{align*}
\text{template } & \text{<int } I\text{> struct X \{} \text{;} } \\
\text{template } & \text{<template <class T> class> struct Z \{} \text{;} } \\
\text{template } & \text{<class T> void f\{typename T::*\}\{} } \\
\text{template } & \text{<class T> void g\{X\::N\:*\}\{} } \\
\text{template } & \text{<class T> void h\{Z\::template TT\:*\}\{} } \\
\text{struct } & \text{A \{} \text{;} \\
\text{struct } & \text{B \{ int Y; \}; } \\
\text{struct } & \text{C \{ } \\
\text{ typedef } & \text{int N; } \\
\text{struct } & \text{D \{ } \\
\text{ typedef } & \text{int TT; } \\
\text{\}; } \\
\text{int main()} \{ \\
\text{ // Deduction fails in each of these cases: } \\
\text{f\{A\}>(0); } & \text{ // A does not contain a member Y} \\
\text{f\{B\}>(0); } & \text{ // The Y member of B is not a type} \\
\text{g\{C\}>(0); } & \text{ // The N member of C is not a non-type} \\
\text{h\{D\}>(0); } & \text{ // The TT member of D is not a template} \\
\text{\}} \\
\end{align*}
\]
— end example

— Attempting to create a pointer to reference type.
— Attempting to create a reference to \texttt{void}.
— Attempting to create “pointer to member of \texttt{T}” when \texttt{T} is not a class type.

[Example 11:]

\[
\begin{align*}
\text{template } & \text{<class T> int f\{int T::*\}\{} } \\
\text{int } & \text{i = f\{int\}>(0);} \\
\end{align*}
\]
— end example

— Attempting to give an invalid type to a non-type template parameter.

[Example 12:]

\[
\begin{align*}
\text{template } & \text{<class T, T> struct S \{} \text{;} } \\
\text{template } & \text{<class T> int f\{S<T, T>()::*\}\{} } \\
\text{struct } & \text{X \{} \text{;} \\
\text{int } & \text{i0 = f\{X\}>(0);} \\
\end{align*}
\]
— end example

— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration.

[Example 13:]

\[
\begin{align*}
\text{template } & \text{<class T, T>*> int f\{int\}; } \\
\text{int } & \text{i2 = f\{int,1\}>(0); } & \text{ // can’t conv 1 to int*} \\
\end{align*}
\]
— end example

— Attempting to create a function type in which a parameter has a type of \texttt{void}, or in which the return type is a function type or array type.

— end note

[Example 14: In the following example, assuming a \texttt{signed char} cannot represent the value 1000, a narrowing conversion (9.4.5) would be required to convert the template-argument of type \texttt{int} \texttt{to signed char}, therefore substitution fails for the second template (13.4.3).]

\[
\begin{align*}
\text{template } & \text{<int> int f\{int\}; } \\
\text{template } & \text{<signed char> int f\{int\}; } \\
\text{int } & \text{i1 = f\{1000\}>(0); } & \text{ // OK} \\
\end{align*}
\]
13.10.3.2  Deducing template arguments from a function call

Template argument deduction is done by comparing each function template parameter type (call it $P$) that contains template-parameters that participate in template argument deduction with the type of the corresponding argument of the call (call it $A$) as described below. If removing references and cv-qualifiers from $P$ gives `std::initializer_list<P'>` or $P'[N]$ for some $P'$ and $N$ and the argument is a non-empty initializer list (9.4.5), then deduction is performed instead for each element of the initializer list independently, taking $P'$ as separate function template parameter types $P'_i$ and the $i^{th}$ initializer element as the corresponding argument. In the $P'[N]$ case, if $N$ is a non-type template parameter, $N$ is deduced from the length of the initializer list. Otherwise, an initializer list argument causes the parameter to be considered a non-deduced context (13.10.3.6).

**Example 1:**

```cpp
template<class T> void f(std::initializer_list<T>);  
f({1,2,3}); // T deduced as int
f({1,"asdf"}); // error: T deduced as both int and const char*

template<class T> void g(T);  
g({1,2,3}); // error: no argument deduced for T

template<class T, int N> void h(T const(&)[N]);  
h({1,2,3}); // T deduced as int; N deduced as 3
j({42}); // T deduced as int; array bound not considered

struct Aggr { int i; int j; };  
template<int N> void k(Aggr const(&)[N]);  
k({1,2,3}); // error: deduction fails, no conversion from int to Aggr
k({{1},{2},{3}}); // OK, N deduced as 3

template<int M, int N> void m(int const(&)[M][N]);  
m({{1,2},{3,4}}); // M and N both deduced as 2

template<class T, int N> void n(T const(&)[N], T);  
n({{1},{2},{3}},Aggr()); // OK, T is Aggr, N is 3

template<typename T, int N> void o(T (* const (&)[N])(T)) { }  
int f1(int);  
int f4(int);  
char f4(char);  
o( &f1, &f4 ); // OK, T deduced as int from first element, nothing
 // deduced from second element, N deduced as 2
o( &f1, static_cast<char(*)(char)>(&f4) ); // error: conflicting deductions for T
```

**Example 2:**

```cpp
template<class ... Types> void f(Types& ...);  
template<class T1, class ... Types> void g(T1, Types ...);  
template<class T1, class ... Types> void g1(Types ..., T1);

void h(int x, float& y) {  
    const int z = x;  
    f(x, y, z); // Types deduced as int, float, const int
    g(x, y, z); // T1 deduced as int; Types deduced as float, int
```
\begin{verbatim}
g1(x, y, z); // error: Types is not deduced
g1<int, int, int>(x, y, z); // OK, no deduction occurs
\end{verbatim}

— end example]

2 If \( P \) is not a reference type:

\begin{enumerate}
\item If \( A \) is an array type, the pointer type produced by the array-to-pointer standard conversion (7.3.3) is used in place of \( A \) for type deduction; otherwise,
\item If \( A \) is a function type, the pointer type produced by the function-to-pointer standard conversion (7.3.4) is used in place of \( A \) for type deduction; otherwise,
\item If \( A \) is a cv-qualified type, the top-level cv-qualifiers of \( A \)'s type are ignored for type deduction.
\end{enumerate}

3 If \( P \) is a cv-qualified type, the top-level cv-qualifiers of \( P \)'s type are ignored for type deduction. If \( P \) is a reference type, the type referred to by \( P \) is used for type deduction.

[Example 3:
\begin{verbatim}
template<class T> int f(const T&);
int n1 = f(5); // calls f<int>(const int&)
const int i = 0;
int n2 = f(i); // calls f<int>(const int&)
template <class T> int g(volatile T&);
int n3 = g(i); // calls g<const int>(const volatile int&)
\end{verbatim}
— end example]

A \textit{forwarding reference} is an rvalue reference to a cv-unqualified template parameter that does not represent a template parameter of a class template (during class template argument deduction (12.4.2.9)). If \( P \) is a forwarding reference and the argument is an lvalue, the type “lvalue reference to \( A \)” is used in place of \( A \) for type deduction.

[Example 4:
\begin{verbatim}
template <class T> int f(T&& heisenreference);
template <class T> int g(const T&&);
int i;
int n1 = f(i); // calls f<int&&>(int&&)
int n2 = f(0); // calls f<int&&>(int&&)
int n3 = g(i); // error: would call g<int>(const int&&), which
// would bind an rvalue reference to an lvalue

template <class T> struct A {
    template <class U>
    A(T&&, U&&, int*);
    // #1: T&& is not a forwarding reference.
    // U&& is a forwarding reference.
    A(T&&, int*); // #2
};

template <class T> A(T&&, int*) -> A<T>; // #3: T&& is a forwarding reference.

int *ip;
A a{i, 0, ip}; // error: cannot deduce from #1
A a0{i, 0, ip}; // uses #1 to deduce A<int> and #1 to initialize
A a2{i, ip}; // uses #3 to deduce A<int&&> and #2 to initialize
\end{verbatim}
— end example]

4 In general, the deduction process attempts to find template argument values that will make the deduced \( A \) identical to \( A \) (after the type \( A \) is transformed as described above). However, there are three cases that allow a difference:

\begin{enumerate}
\item If the original \( P \) is a reference type, the deduced \( A \) (i.e., the type referred to by the reference) can be more cv-qualified than the transformed \( A \).
\item The transformed \( A \) can be another pointer or pointer-to-member type that can be converted to the deduced \( A \) via a function pointer conversion (7.3.14) and/or qualification conversion (7.3.6).
\end{enumerate}
If \( P \) is a class and \( P \) has the form `simple-template-id`, then the transformed \( A \) can be a derived class \( D \) of the deduced \( A \). Likewise, if \( P \) is a pointer to a class of the form `simple-template-id`, the transformed \( A \) can be a pointer to a derived class \( D \) pointed to by the deduced \( A \). However, if there is a class \( C \) that is a (direct or indirect) base class of \( D \) and derived (directly or indirectly) from a class \( B \) and that would be a valid deduced \( A \), the deduced \( A \) cannot be \( B \) or pointer to \( B \), respectively.

**Example 5:**
```
template <typename... T> struct X;
template <> struct X<> {};  // calls f<int>, not f<>
  // B is X<>, C is X<int>
template <typename T, typename... Ts> struct X<T, Ts...> : X<Ts...> {};  // calls f<int>, not f<>
```
```
struct D : X<int> {};  // calls f<int>, not f<>
struct E : X<>, X<int> {};  // calls f<int>, not f<>
```
```
int f(const X<T...>&);  // calls f<int>, not f<>  
```
```
int x = f(D());  // calls f<int>, not f<>
int z = f(E());  // calls f<int>, not f<>
```

**Example 6:**
```
// Only one function of an overload set matches the call so the function parameter is a deduced context.
template <class T> int f(T (*p)(T));
int g(int);
int g(char);
int i = f(g);  // calls f(int (*)(int))
```

**Example 7:**
```
// Ambiguous deduction causes the second function parameter to be a non-deduced context.
template <class T> int f(T, T (*p)(T));
int g(int);
char g(char);
int i = f(1, g);  // calls f(int, int (*)(int))
```

**Example 8:**
```
// The overload set contains a template, causing the second function parameter to be a non-deduced context.
template <class T> int f(T, T (*p)(T));
char g(char);
template <class T> T g(T);
int i = f(1, g);  // calls f(int, int (*)(int))
```

These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced \( A \), the type deduction fails.

**Note 1:** If a template-parameter is not used in any of the function parameters of a function template, or is used only in a non-deduced context, its corresponding template-argument cannot be deduced from a function call and the template-argument must be explicitly specified. —end note

When \( P \) is a function type, function pointer type, or pointer-to-member-function type:

- If the argument is an overload set containing one or more function templates, the parameter is treated as a non-deduced context.
- If the argument is an overload set (not containing function templates), trial argument deduction is attempted using each of the members of the set. If deduction succeeds for only one of the overload set members, that member is used as the argument value for the deduction. If deduction succeeds for more than one member of the overload set the parameter is treated as a non-deduced context.

**Example 6:**
```
// Only one function of an overload set matches the call so the function parameter is a deduced context.
template <class T> int f(T (*p)(T));
int g(int);
int g(char);
int i = f(g);  // calls f(int (*)(int))
```

**Example 7:**
```
// Ambiguous deduction causes the second function parameter to be a non-deduced context.
template <class T> int f(T, T (*p)(T));
int g(int);
char g(char);
int i = f(1, g);  // calls f(int, int (*)(int))
```

**Example 8:**
```
// The overload set contains a template, causing the second function parameter to be a non-deduced context.
template <class T> int f(T, T (*p)(T));
char g(char);
template <class T> T g(T);
int i = f(1, g);  // calls f(int, int (*)(int))
```

If deduction succeeds for all parameters that contain template-parameters that participate in template argument deduction, and all template arguments are explicitly specified, deduced, or obtained from default template arguments, remaining parameters are then compared with the corresponding arguments. For each
remaining parameter \( P \) with a type that was non-dependent before substitution of any explicitly-specified template arguments, if the corresponding argument \( A \) cannot be implicitly converted to \( P \), deduction fails.

[Note 2: Parameters with dependent types in which no template-parameters participate in template argument deduction, and parameters that became non-dependent due to substitution of explicitly-specified template arguments, will be checked during overload resolution. — end note]

[Example 9:

```cpp
template <class T> struct Z {
    typedef typename T::x xx;
};
template <class T> typename Z<T>::xx f(void *, T); // #1
template <class T> void f(int, T); // #2
struct A {} a;
int main() {
    f(1, a);    // OK, deduction fails for #1 because there is no conversion from int to void*
}
```
— end example]

13.10.3.3 Deducing template arguments taking the address of a function template
[ temp.deduct.funcaddr ]

1 Template arguments can be deduced from the type specified when taking the address of an overloaded function (12.5). If there is a target, the function template’s function type and the target type are used as the types of \( P \) and \( A \), and the deduction is done as described in 13.10.3.6. Otherwise, deduction is performed with empty sets of types \( P \) and \( A \).

2 A placeholder type (9.2.9.6) in the return type of a function template is a non-deduced context. If template argument deduction succeeds for such a function, the return type is determined from instantiation of the function body.

13.10.3.4 Deducing conversion function template arguments
[ temp.deduct.conv ]

1 Template argument deduction is done by comparing the return type of the conversion function template (call it \( P \)) with the type that is required as the result of the conversion (call it \( A \); see 9.4, 12.4.2.6, and 12.4.2.7 for the determination of that type) as described in 13.10.3.6.

2 If \( P \) is a reference type, the type referred to by \( P \) is used in place of \( P \) for type deduction and for any further references to or transformations of \( P \) in the remainder of this subclause.

3 If \( A \) is not a reference type:

   (3.1) — If \( P \) is an array type, the pointer type produced by the array-to-pointer standard conversion (7.3.3) is used in place of \( P \) for type deduction; otherwise,

   (3.2) — If \( P \) is a function type, the pointer type produced by the function-to-pointer standard conversion (7.3.4) is used in place of \( P \) for type deduction; otherwise,

   (3.3) — If \( P \) is a cv-qualified type, the top-level cv-qualifiers of \( P \)’s type are ignored for type deduction.

4 If \( A \) is a cv-qualified type, the top-level cv-qualifiers of \( A \)’s type are ignored for type deduction. If \( A \) is a reference type, the type referred to by \( A \) is used for type deduction.

5 In general, the deduction process attempts to find template argument values that will make the deduced \( A \) identical to \( A \). However, there are four cases that allow a difference:

   (5.1) — If the original \( A \) is a reference type, \( A \) can be more cv-qualified than the deduced \( A \) (i.e., the type referred to by the reference).

   (5.2) — If the original \( A \) is a function pointer type, \( A \) can be “pointer to function” even if the deduced \( A \) is “pointer to noexcept function”.

   (5.3) — If the original \( A \) is a pointer-to-member-function type, \( A \) can be “pointer to member of type function” even if the deduced \( A \) is “pointer to member of type noexcept function”.

   (5.4) — The deduced \( A \) can be another pointer or pointer-to-member type that can be converted to \( A \) via a qualification conversion.

6 These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced \( A \), the type deduction fails.
Template argument deduction is done by comparing certain types associated with the two function templates being compared.

Two sets of types are used to determine the partial ordering. For each of the templates involved there is the original function type and the transformed function type.

[Note 1: The creation of the transformed type is described in 13.7.7.3. — end note]

The deduction process uses the transformed type as the argument template and the original type of the other template as the parameter template. This process is done twice for each type involved in the partial ordering comparison: once using the transformed template-1 as the argument template and template-2 as the parameter template and again using the transformed template-2 as the argument template and template-1 as the parameter template.

The types used to determine the ordering depend on the context in which the partial ordering is done:

- In the context of a function call, the types used are those function parameter types for which the function call has arguments.
- In the context of a call to a conversion function, the return types of the conversion function templates are used.
- In other contexts (13.7.7.3) the function template’s function type is used.

Each type nominated above from the parameter template and the corresponding type from the argument template are used as the types of \( P \) and \( A \).

Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:

- If \( P \) is a reference type, \( P \) is replaced by the type referred to.
- If \( A \) is a reference type, \( A \) is replaced by the type referred to.
- If both \( P \) and \( A \) were reference types (before being replaced with the type referred to above), determine which of the two types (if any) is more cv-qualified than the other; otherwise the types are considered to be equally cv-qualified for partial ordering purposes. The result of this determination will be used below.

Remove any top-level cv-qualifiers:

- If \( P \) is a cv-qualified type, \( P \) is replaced by the cv-unqualified version of \( P \).
- If \( A \) is a cv-qualified type, \( A \) is replaced by the cv-unqualified version of \( A \).

Using the resulting types \( P \) and \( A \), the deduction is then done as described in 13.10.3.6. If \( P \) is a function parameter pack, the type \( A \) of each remaining parameter type of the argument template is compared with the type \( P \) of the declarator-id of the function parameter pack. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. Similarly, if \( A \) was transformed from a function parameter pack, it is compared with each remaining parameter type of the parameter template. If deduction succeeds for a given type, the type from the argument template is considered to be at least as specialized as the type from the parameter template.

[Example 1:

```cpp
template<class... Args> void f(Args... args); // #1
template<class T1, class... Args> void f(T1 a1, Args... args); // #2
template<class T1, class T2> void f(T1 a1, T2 a2); // #3

f(); // calls #1
f(1, 2, 3); // calls #2
f(1, 2); // calls #3; non-variadic template #3 is more specialized
          // than the variadic templates #1 and #2
@end example]

If, for a given type, the types are identical after the transformations above and both \( P \) and \( A \) were reference types (before being replaced with the type referred to above):

---

\(143\) Default arguments are not considered to be arguments in this context; they only become arguments after a function has been selected.

§ 13.10.3.5
if the type from the argument template was an lvalue reference and the type from the parameter template was not, the parameter type is not considered to be at least as specialized as the argument type; otherwise,

if the type from the argument template is more cv-qualified than the type from the parameter template (as described above), the parameter type is not considered to be at least as specialized as the argument type.

Function template \( F \) is at least as specialized as function template \( G \) if, for each pair of types used to determine the ordering, the type from \( F \) is at least as specialized as the type from \( G \). \( F \) is more specialized than \( G \) if \( F \) is at least as specialized as \( G \) and \( G \) is not at least as specialized as \( F \).

If, after considering the above, function template \( F \) is at least as specialized as function template \( G \) and vice-versa, and if \( G \) has a trailing function parameter pack for which \( F \) does not have a corresponding parameter, and if \( F \) does not have a trailing function parameter pack, then \( F \) is more specialized than \( G \).

In most cases, deduction fails if not all template parameters have values, but for partial ordering purposes a template parameter may remain without a value provided it is not used in the types being used for partial ordering.

[Note 2: A template parameter used in a non-deduced context is considered used. — end note]

[Example 2:]
```cpp
template <class T> T f(int); // #1
template <class T, class U> T f(U); // #2
void g() {
    f<int>(1); // calls #1
}
```
—end example]

[Note 3: Partial ordering of function templates containing template parameter packs is independent of the number of deduced arguments for those template parameter packs. — end note]

[Example 3:]
```cpp
template<class ...> struct Tuple { };
template<class ... Types> void g(Tuple<Types ...>);  // #1

template<class T1, class ... Types> void g(Tuple<T1, Types ...>);  // #2

template<class T1, class ... Types> void g(Tuple<T1, Types& ...>); // #3

g(Tuple<>());  // calls #1

g(Tuple<int, float>());  // calls #2

g(Tuple<int, float&>()); // calls #3

g(Tuple<int>());  // calls #3
```
—end example]

### 13.10.3.6 Deducing template arguments from a type

Template arguments can be deduced in several different contexts, but in each case a type that is specified in terms of template parameters (call it \( P \)) is compared with an actual type (call it \( A \)), and an attempt is made to find template argument values (a type for a type parameter, a value for a non-type parameter, or a template for a template parameter) that will make \( P \), after substitution of the deduced values (call it the deduced \( A \)), compatible with \( A \).

In some cases, the deduction is done using a single set of types \( P \) and \( A \), in other cases, there will be a set of corresponding types \( P \) and \( A \). Type deduction is done independently for each \( P/A \) pair, and the deduced template argument values are then combined. If type deduction cannot be done for any \( P/A \) pair, or if for any pair the deduction leads to more than one possible set of deduced values, or if different pairs yield different deduced values, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails. The type of a type parameter is only deduced from an array bound if it is not otherwise deduced.

A given type \( P \) can be composed from a number of other types, templates, and non-type values:

- A function type includes the types of each of the function parameters and the return type.
- A pointer-to-member type includes the type of the class object pointed to and the type of the member pointed to.

§ 13.10.3.6
— A type that is a specialization of a class template (e.g., `A<int>`) includes the types, templates, and non-type values referenced by the template argument list of the specialization.

— An array type includes the array element type and the value of the array bound.

In most cases, the types, templates, and non-type values that are used to compose \( P \) participate in template argument deduction. That is, they may be used to determine the value of a template argument, and template argument deduction fails if the value so determined is not consistent with the values determined elsewhere. In certain contexts, however, the value does not participate in type deduction, but instead uses the values of template arguments that were either deduced elsewhere or explicitly specified. If a template parameter is used only in non-deduced contexts and is not explicitly specified, template argument deduction fails.

[Note 1: Under 13.10.3.2, if \( P \) contains no template-parameters that appear in deduced contexts, no deduction is done, so \( P \) and \( A \) need not have the same form. — end note]

The non-deduced contexts are:

1. The nested-name-specifier of a type that was specified using a qualified-id.
2. The expression of a decltype-specifier.
3. A non-type template argument or an array bound in which a subexpression references a template parameter.
4. A template parameter used in the parameter type of a function parameter that has a default argument that is being used in the call for which argument deduction is being done.
5. A function parameter for which the associated argument is an overload set (12.5), and one or more of the following apply:
   1. more than one function matches the function parameter type (resulting in an ambiguous deduction), or
   2. no function matches the function parameter type, or
   3. the overload set supplied as an argument contains one or more function templates.
6. A function parameter for which the associated argument is an initializer list (9.4.5) but the parameter does not have a type for which deduction from an initializer list is specified (13.10.3.2).

[Example 1:
```c++
template<class T> void g(T);
g({1,2,3});  // error: no argument deduced for T
```
— end example]

— A function parameter pack that does not occur at the end of the parameter-declaration-list.

When a type name is specified in a way that includes a non-deduced context, all of the types that comprise that type name are also non-deduced. However, a compound type can include both deduced and non-deduced types.

[Example 2: If a type is specified as \( A<T>:B<T2> \), both \( T \) and \( T2 \) are non-deduced. Likewise, if a type is specified as \( A<T>:B<T>, I, J, \) and \( T \) are non-deduced. If a type is specified as \( void f(typename A<T>:B, A<T>) \), the \( T \) in \( A<T>:B \) is non-deduced but the \( T \) in \( A<T> \) is deduced. — end example]

[Example 3: Here is an example in which different parameter/argument pairs produce inconsistent template argument deductions:
```c++
template<class T> void f(T x, T y) { /* ... */ }
struct A { /* ... */ };
struct B : A { /* ... */ };
void g(A a, B b) {
  f(a,b);  // error: T could be A or B
  f(b,a);  // error: T could be A or B
  f(a,a);  // OK: T is A
  f(b,b);  // OK: T is B
}
```
Here is an example where two template arguments are deduced from a single function parameter/argument pair. This can lead to conflicts that cause type deduction to fail:
```c++
template <class T, class U> void f(T (*)(T, U, U));
```
```c
int g1( int, float, float);
char g2( int, float, float);
int g3( int, char, float);

void r() {
f(g1);  // OK: T is int and U is float
f(g2);  // error: T could be char or int
f(g3);  // error: U could be char or float
}
```

Here is an example where a qualification conversion applies between the argument type on the function call and the deduced template argument type:

```c
template<class T> void f(const T*) { }
int* p;
void s() {
f(p);  // f(const int*)
}
```

Here is an example where the template argument is used to instantiate a derived class type of the corresponding function parameter type:

```c
template <class T> struct B { };
template <class T> struct D : public B<T> {}; struct D2 : public B<int> {}; template <class T> void f(B<T>&){}
void t() {
  D<int> d;
  D2 d2;
  f(d);  // calls f(B<int>&)
  f(d2);  // calls f(B<int>&)
}
```

A template type argument T, a template template argument TT or a template non-type argument i can be deduced if P and A have one of the following forms:

- T
- cv T
- T*
- T&
- T&&
- T[integer-constant]

- template-name<T> (where template-name refers to a class template)

- type(T)
- T()
- T type::*
- type T::*
- T T::*
- T (type::*())
- type (T::*())()
- type (type::*)(T)
- type (T::*)(T)
- T (type::*)(T)
- T (T::*)(T)
- T (T::*)(T)

- template-name<i> (where template-name refers to a class template)

- TT<T>
- TT<i>
- TT<>

where (T) represents a parameter-type-list (9.3.4.6) where at least one parameter type contains a T, and () represents a parameter-type-list where no parameter type contains a T. Similarly, <T> represents template argument lists where at least one argument contains a T, <i> represents template argument lists where at
least one argument contains an i and <> represents template argument lists where no argument contains a T or an i.

9 If P has a form that contains <T> or <i>, then each argument P_i of the respective template argument list of P is compared with the corresponding argument A_i of the corresponding template argument list of A. If the template argument list of P contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If P_i is a pack expansion, then the pattern of P_i is compared with each remaining argument in the template argument list of A. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by P_i. During partial ordering (13.10.3.5), if A_i was originally a pack expansion:

(9.1) — if P does not contain a template argument corresponding to A_i then A_i is ignored;
(9.2) — otherwise, if P_i is not a pack expansion, template argument deduction fails.

[Example 4]:

```c
template<class T1, class... Z> class S; // #1
template<class T1, class... Z> class S<T1, const Z&...> { }; // #2
template<class T1, class T2> class S<T1, const T2&> { }; // #3
S<int, const int&> s; // both #2 and #3 match; #3 is more specialized
```

...end example]

10 Similarly, if P has a form that contains (T), then each parameter type P_i of the respective parameter-type-list (9.3.4.6) of P is compared with the corresponding parameter type A_i of the corresponding parameter-type-list of A. If P and A are function types that originated from deduction when taking the address of a function template (13.10.3.3) or when deducing template arguments from a function declaration (13.10.3.7) and P_i and A_i are parameters of the top-level parameter-type-list of P and A, respectively, P_i is adjusted if it is a forwarding reference (13.10.3.2) and A_i is an lvalue reference, in which case the type of P_i is changed to be the template parameter type (i.e., T&& is changed to simply T).

[Note 2: As a result, when P_i is T&& and A_i is X&, the adjusted P_i will be T, causing T to be deduced as X&. — end note]

[Example 5]:

```c
template <class T> void f(T&&);
template <> void f(int&) { } // #1
template <> void f(int&&) { } // #2
void g(int i) {
  f(i); // calls f<int&&>(int&&), i.e., #1
  f(0); // calls f<int>(int&&), i.e., #2
}
```

...end example]

If the parameter-declaration corresponding to P_i is a function parameter pack, then the type of its declarator-id is compared with each remaining parameter type in the parameter-type-list of A. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. During partial ordering (13.10.3.5), if A_i was originally a function parameter pack:

(10.1) — if P does not contain a function parameter type corresponding to A_i then A_i is ignored;
(10.2) — otherwise, if P_i is not a function parameter pack, template argument deduction fails.

[Example 6]:

```c
template<class T, class... U> void f(T*, U...) { } // #1
template<class T> void f(T) { } // #2
template void f(int*); // selects #1
```

...end example]

11 These forms can be used in the same way as T is for further composition of types.

[Example 7]:

```c
X<int> (*)(char[6])
```
is of the form

\texttt{template-name}\langle T \rangle \ (\ast) (\text{\texttt{type}[i]})

which is a variant of

\texttt{type} \ (\ast) (T)

where \texttt{type} is \texttt{X<int>} and \texttt{T} is \texttt{char[6]}. —end example

12 Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

13 When the value of the argument corresponding to a non-type template parameter \(P\) that is declared with a dependent type is deduced from an expression, the template parameters in the type of \(P\) are deduced from the type of the value.

[Example 8:

\begin{verbatim}

    template<long n> struct A {
    
    template<typename T> struct C;
    template<typename T, T n> struct C<A<n>> {
        using Q = T;
    
    using R = long;
    using R = C<A<2>>::Q; // OK; \(T\) was deduced as \texttt{long} from the
        \hspace{10cm} \text{template argument value in the type \texttt{A<2>}}
    
    \end{verbatim}

—end example]

14 The type of \(N\) in the type \(T[N]\) is \texttt{std::size_t}.

[Example 9:

\begin{verbatim}

    template<typename T> struct S;
    template<typename T, T n> struct S<int[n]> {
        using Q = T;
    
    using V = decltype(sizeof 0);
    using V = S<int[42>>::Q; // OK; \(T\) was deduced as \texttt{std::size_t} from the type \texttt{int[42]}
    
    \end{verbatim}

—end example]

15 [Example 10:

\begin{verbatim}

    template<class T, T i> void f(int (&a)[i]);
    int v[10];
    void g() {
        f(v); // OK: \(T\) is \texttt{std::size_t}
    
    \end{verbatim}

—end example]

16 [Note 3: Except for reference and pointer types, a major array bound is not part of a function parameter type and cannot be deduced from an argument:

\begin{verbatim}

    template<int i> void f1(int a[10][i]);
    template<int i> void f2(int a[i][20]);
    template<int i> void f3(int (&a)[i][20]);

    void g() {
        int v[10][20];
        f1(v); // OK: \(i\) deduced as 20
        f1<20>(v); // OK
        f2(v); // error: cannot deduce template-argument \(i\)
        f2<10>(v); // OK
        f3(v); // OK: \(i\) deduced as 10
    
    \end{verbatim}

—end note]
[Note 4: If, in the declaration of a function template with a non-type template parameter, the non-type template parameter is used in a subexpression in the function parameter list, the expression is a non-deduced context as specified above.]

Example 11:

```cpp
template <int i> class A { /* ... */
    template <int i> void g(A<i+1>);
    template <int i> void f(A<i>, A<i+1>);

    void k() {
        A<i> a1;
        A<i+2> a2;
        g(a1);               // error: deduction fails for expression i+1
        g<0>(a1);           // OK
        f(a1, a2);          // OK
    }

    — end example

    — end note
```

[Note 5: Template parameters do not participate in template argument deduction if they are used only in non-deduced contexts. For example,]

```cpp
template<int i, typename T>
T deduce(typename A<T>::X x, // T is not deduced here
         T t,               // but T is deduced here
         typename B<i>::Y y); // i is not deduced here

A<int> a;
B<77> b;

int x = deduce<77>(a.xm, 62, b.ym);
    // T deduced as int; a.xm must be convertible to A<int>::X
    // i is explicitly specified to be 77; b.ym must be convertible to B<77>::Y

— end note
```

18 If \( P \) has a form that contains \(<i>\), and if the type of \( i \) differs from the type of the corresponding template parameter of the template named by the enclosing simple-template-id, deduction fails. If \( P \) has a form that contains \([i]\), and if the type of \( i \) is not an integral type, deduction fails.\footnote{Although the template-argument corresponding to a template-parameter of type \texttt{bool} can be deduced from an array bound, the resulting value will always be \texttt{true} because the array bound will be nonzero.}

Example 12:

```cpp
template<int i> class A { /* ... */
    template<short s> void f(A<s>);

    void k1() {
        A<i> a;
        f(a);               // error: deduction fails for conversion from int to short
        f<1>(a);           // OK
    }

    template<const short cs> class B { }
    template<short s> void g(B<s>);

    void k2() {
        B<i> b;
        g(b);               // OK: cv-qualifiers are ignored on template parameter types
    }

    — end example
```

A template-argument can be deduced from a function, pointer to function, or pointer-to-member-function type.

Example 13:

```cpp
template<class T> void f(void(*)(T,int));
```

§ 13.10.3.6
21 A template type-parameter cannot be deduced from the type of a function default argument.

[Example 14]:

```cpp
template <class T> void f(T = 5, T = 7);
void g() {
    f(1); // OK: call f<int>(1,7)
    f(); // error: cannot deduce T
    f<int>(); // OK: call f<int>(5,7)
}
```

—end example]

22 The template-argument corresponding to a template template-parameter is deduced from the type of the template-argument of a class template specialization used in the argument list of a function call.

[Example 15]:

```cpp
template <template <class T> class X> struct A { };
template <template <class T> class X> void f(A<X>) { }
template<class T> struct B { };
A<B> ab;
f(ab); // calls f(A<B>)
```

—end example]

[Note 6: Template argument deduction involving parameter packs (13.7.4) can deduce zero or more arguments for each parameter pack. —end note]

[Example 16]:

```cpp
template<class> struct X { };
template<class R, class ... ArgTypes> struct X<R(int, ArgTypes ...)> { };
template<class ... Types> struct Y { };
template<class T, class ... Types> struct Y<T, Types ...> { };

template<class ... Types> int f(void (*)(Types ...));
void g(int, float);

X<int> x1; // uses primary template
X<int(int, float, double)> x2; // uses partial specialization; ArgTypes contains float, double
X<int(float, int)> x3; // uses primary template
Y<int> y1; // use primary template; Types is empty
Y<int&, float&, double&> y2; // uses partial specialization; T is int&, Types contains float, double
Y<int, float, double> y3; // uses primary template; Types contains int, float, double
int fv = f(g); // OK; Types contains int, float
```

—end example]

13.10.3.7 Deducing template arguments from a function declaration [temp.deduct.decl]

In a declaration whose declarator-id refers to a specialization of a function template, template argument deduction is performed to identify the specialization to which the declaration refers. Specifically, this is done for explicit instantiations (13.9.3), explicit specializations (13.9.4), and certain friend declarations (13.7.5). This is also done to determine whether a deallocation function template specialization matches a placement operator new (6.7.5.5.3, 7.6.2.8). In all these cases, P is the type of the function template being considered as a potential match and A is either the function type from the declaration or the type of the deallocation function that would match the placement operator new as described in 7.6.2.8. The deduction is done as described in 13.10.3.6.
If, for the set of function templates so considered, there is either no match or more than one match after partial ordering has been considered (13.7.7.3), deduction fails and, in the declaration cases, the program is ill-formed.

13.10.4 Overload resolution

When a call to the name of a function or function template is written (explicitly, or implicitly using the operator notation), template argument deduction (13.10.3) and checking of any explicit template arguments (13.4) are performed for each function template to find the template argument values (if any) that can be used with that function template to instantiate a function template specialization that can be invoked with the call arguments. For each function template, if the argument deduction and checking succeeds, the template-arguments (deduced and/or explicit) are used to synthesize the declaration of a single function template specialization which is added to the candidate functions set to be used in overload resolution. If, for a given function template, argument deduction fails or the synthesized function template specialization would be ill-formed, no such function is added to the set of candidate functions for that template. The complete set of candidate functions includes all the synthesized declarations and all of the non-template overloaded functions of the same name. The synthesized declarations are treated like any other functions in the remainder of overload resolution, except as explicitly noted in 12.4.4.\footnote{145}

\footnote{145}The parameters of function template specializations contain no template parameter types. The set of conversions allowed on deduced arguments is limited, because the argument deduction process produces function templates with parameters that either match the call arguments exactly or differ only in ways that can be bridged by the allowed limited conversions. Non-deduced arguments allow the full range of conversions. Note also that 12.4.4 specifies that a non-template function will be given preference over a template specialization if the two functions are otherwise equally good candidates for an overload match.}

Example 1:

```cpp
template<class T> T max(T a, T b) { return a>b?a:b; }

void f(int a, int b, char c, char d) {
    int m1 = max(a,b); // max(int, int)
    char m2 = max(c,d); // max(char, char)
    int m3 = max(a,c);  // error: cannot generate max(int, char)
}
```

Adding the non-template function

```cpp
int max(int, int);
```
to the example above would resolve the third call, by providing a function that could be called for `max(a, c)` after using the standard conversion of `char` to `int` for `c`. — end example]

Example 2: Here is an example involving conversions on a function argument involved in `template-argument` deduction:

```cpp
template<class T> struct B { /* ... */};

void h(int*, int i, char c) {
    f(pi,i);  // #1: `f<int*>(pi,i)`
    f(pi,c);  // #2: `f<int*>(pi,c)`
    f(i,c);   // #2: `f<int>(i,c)`
}  // end example
```

Example 3: Here is an example involving conversions on a function argument not involved in `template-parameter` deduction:

```cpp
void h(int* pi, int i, char c) {
    f(pi,i);  // #1: `f<int>(pi,i)`
    f(pi,c);  // #2: `f<int*>(pi,c)`
    f(i,c);   // #2: `f<int>(i,c)`
    f(i,i);   // #2: `f<int>(i,char(i))`
}  // end example
```
Only the signature of a function template specialization is needed to enter the specialization in a set of candidate functions. Therefore only the function template declaration is needed to resolve a call for which a template specialization is a candidate.

Example 4:

```c
template<class T> void f(T); // declaration

void g() {
    f("Annemarie"); // call of f<const char*>
}
```

The call of `f` is well-formed even if the template `f` is only declared and not defined at the point of the call. The program will be ill-formed unless a specialization for `f<const char*>` is explicitly instantiated in some translation unit (13.1). — end example]
14 Exception handling

14.1 Preamble

Exception handling provides a way of transferring control and information from a point in the execution of a thread to an exception handler associated with a point previously passed by the execution. A handler will be invoked only by throwing an exception in code executed in the handler’s try block or in functions called from the handler’s try block.

```
try-block:
  try compound-statement handler-seq
function-try-block:
  try ctor-initializer Opt compound-statement handler-seq
handler-seq:
  handler handler-seq Opt
handler:
  catch ( exception-declaration ) compound-statement
exception-declaration:
  attribute-specifier-seq Opt type-specifier-seq declarator
  attribute-specifier-seq Opt type-specifier-seq abstract-declarator Opt...
```

The optional attribute-specifier-seq in an exception-declaration appertains to the parameter of the catch clause (14.4).

2 A try-block is a statement (8.1).

[Note 1: Within this Clause “try block” is taken to mean both try-block and function-try-block. —end note]

3 A goto or switch statement shall not be used to transfer control into a try block or into a handler.

[Example 1:

```c
void f() {
  goto l1; // error
  goto l2; // error
  try {
    goto l1; // OK
    goto l2; // error
  }
  catch (...) {
    l1: ;
    goto l1; // error
    goto l2; // OK
  }
}
```

—end example]

A goto, break, return, or continue statement can be used to transfer control out of a try block or handler. When this happens, each variable declared in the try block will be destroyed in the context that directly contains its declaration.

[Example 2:

```c
lab: try {
  T1 t1;
  try {
    T2 t2;
    if (condition)
      goto lab;
  } catch(...) { /* handler 2 */ }
  catch(...) { /* handler 1 */ }
}
```

§ 14.1
Here, executing `goto lab;` will destroy first `t2`, then `t1`, assuming the condition does not declare a variable. Any exception thrown while destroying `t2` will result in executing handler 2; any exception thrown while destroying `t1` will result in executing handler 1. —end example

4 A function-try-block associates a handler-seq with the ctor-initializer, if present, and the compound-statement. An exception thrown during the execution of the compound-statement or, for constructors and destructors, during the initialization or destruction, respectively, of the class’s subobjects, transfers control to a handler in a function-try-block in the same way as an exception thrown during the execution of a try-block transfers control to other handlers.

[Example 3:]
```
int f(int);
class C {
  int i;
  double d;
public:
  C(int, double);
};
C::C(int ii, double id)
try : i(f(ii)), d(id) {
  // constructor statements
} catch (...) {
  // handles exceptions thrown from the ctor-initializer and from the constructor statements
}
—end example]

5 In this Clause, “before” and “after” refer to the “sequenced before” relation (6.9.1).

14.2 Throwing an exception [except.throw]

1 Throwing an exception transfers control to a handler.

[Note 1: An exception can be thrown from one of the following contexts: throw-expressions (7.6.18), allocation functions (6.7.5.5.2), dynamic_cast (7.6.1.7), typeid (7.6.1.8), new-expressions (7.6.2.8), and standard library functions (16.3.2.4). —end note]

An object is passed and the type of that object determines which handlers can catch it.

[Example 1:]
```
throw "Help!";
```
can be caught by a handler of const char* type:
```
try {
  // ...
} catch(const char* p) {
  // handle character string exceptions here
}
```

and
```
class Overflow {
  public:
    Overflow(char, double, double);
};
```
```
void f(double x) {
  throw Overflow('+', x, 3.45e107);
}
```
can be caught by a handler for exceptions of type Overflow:
```
try {
  f(1.2);
} catch(Overflow& oo) {
  // handle exceptions of type Overflow here
}
—end example]
When an exception is thrown, control is transferred to the nearest handler with a matching type (14.4); “nearest” means the handler for which the compound-statement or ctor-initializer following the try keyword was most recently entered by the thread of control and not yet exited.

Throwing an exception copy-initializes (9.4, 11.4.5.3) a temporary object, called the exception object. An lvalue denoting the temporary is used to initialize the variable declared in the matching handler (14.4). If the type of the exception object would be an incomplete type, an abstract class type (11.7.4), or a pointer to an incomplete type other than cv void the program is ill-formed.

The memory for the exception object is allocated in an unspecified way, except as noted in 6.7.5.5.2. If a handler exits by rethrowing, control is passed to another handler for the same exception object. The points of potential destruction for the exception object are:

- when an active handler for the exception exits by any means other than rethrowing, immediately after the destruction of the object (if any) declared in the exception-declaration in the handler;
- when an object of type std::exception_ptr (17.9.7) that refers to the exception object is destroyed, before the destructor of std::exception_ptr returns.

Among all points of potential destruction for the exception object, there is an unspecified last one where the exception object is destroyed. All other points happen before that last one (6.9.2.2).

The implementation may then deallocate the memory for the exception object; any such deallocation is done in an unspecified way.

When the thrown object is a class object, the constructor selected for the copy-initialization as well as the constructor selected for a copy-initialization considering the thrown object as an lvalue shall be non-deleted and accessible, even if the copy/move operation is elided (11.10.6). The destructor is potentially invoked (11.4.7).

An exception is considered caught when a handler for that exception becomes active (14.4). An exception can have active handlers and still be considered uncaught if it is rethrown.

If the exception handling mechanism handling an uncaught exception (14.6.3) directly invokes a function that exits via an exception, the function std::terminate is called (14.6.2).

```c
struct C {
    C() { } // throw during copy to handler’s exception-declaration object (14.4)
    C(const C&) { if (std::uncaught_exceptions()) { throw 0; } } // called std::terminate if construction of the handler’s exception-declaration object is not elided (11.10.6)
};

int main() {
    try {
        throw C(); // calls std::terminate if construction of the handler’s exception-declaration object is not elided (11.10.6)
    } catch(C) { }
}
```

A destructor directly invoked by stack unwinding exits via an exception, std::terminate is invoked.

The exception handling mechanism handles an uncaught exception (14.6.3) directly invokes a function that exits via an exception, the function std::terminate is called (14.6.2).

14.3 Constructors and destructors

As control passes from the point where an exception is thrown to a handler, objects with automatic storage duration are destroyed by a process, specified in this subclause, called stack unwinding.

Each object with automatic storage duration is destroyed if it has been constructed, but not yet destroyed, since the try block was entered. If an exception is thrown during the destruction of temporaries or local
variables for a `return` statement (8.7.4), the destructor for the returned object (if any) is also invoked. The objects are destroyed in the reverse order of the completion of their construction.

[Example 1:]
```cpp
struct A { };

struct Y { ~Y() noexcept(false) { throw 0; } };

A f() {
  try {
    A a;
    Y y;
    A b;
    return {}; // #1
  } catch (...) {
  }
  return {}; // #2
}
```

At #1, the returned object of type `A` is constructed. Then, the local variable `b` is destroyed (8.7). Next, the local variable `y` is destroyed, causing stack unwinding, resulting in the destruction of the returned object, followed by the destruction of the local variable `a`. Finally, the returned object is constructed again at #2. — end example]

3 If the initialization or destruction of an object other than by delegating constructor is terminated by an exception, the destructor is invoked for each of the object’s direct subobjects and, for a complete object, virtual base class subobjects, whose initialization has completed (9.4) and whose destructor has not yet begun execution, except that in the case of destruction, the variant members of a union-like class are not destroyed.

[Note 1: If such an object has a reference member that extends the lifetime of a temporary object, this ends the lifetime of the reference member, so the lifetime of the temporary object is effectively not extended. — end note]

The subobjects are destroyed in the reverse order of the completion of their construction. Such destruction is sequenced before entering a handler of the function-try-block of the constructor or destructor, if any.

4 If the `compound-statement` of the function-body of a delegating constructor for an object exits via an exception, the object’s destructor is invoked. Such destruction is sequenced before entering a handler of the function-try-block of a delegating constructor for that object, if any.

[Note 2: If the object was allocated by a `new-expression` (7.6.2.8), the matching deallocation function (6.7.5.5.3), if any, is called to free the storage occupied by the object. — end note]

### 14.4 Handling an exception [except.handle]

1 The `exception-declaration` in a handler describes the type(s) of exceptions that can cause that handler to be entered. The `exception-declaration` shall not denote an incomplete type, an abstract class type, or an rvalue reference type. The `exception-declaration` shall not denote a pointer or reference to an incomplete type, other than “pointer to `cv void`”.

2 A handler of type “array of `T`” or function type `T` is adjusted to be of type “pointer to `T`”.

3 A handler is a match for an exception object of type `E` if

1. The `handler` is of type `cv T` or `cv T&` and `E` and `T` are the same type (ignoring the top-level `cv-qualifiers`), or
2. the `handler` is of type `cv T` or `cv T&` and `T` is an unambiguous public base class of `E`, or
3. the `handler` is of type `cv T` or `const T` where `T` is a pointer or pointer-to-member type and `E` is a pointer or pointer-to-member type that can be converted to `T` by one or more of
   1. a standard pointer conversion (7.3.12) not involving conversions to pointers to private or protected or ambiguous classes
   2. a function pointer conversion (7.3.14)
   3. a qualification conversion (7.3.6), or
4. the `handler` is of type `cv T` or `const T&` where `T` is a pointer or pointer-to-member type and `E` is `std::nullptr_t`.

§ 14.4 455
[Note 1: A throw-expression whose operand is an integer literal with value zero does not match a handler of pointer or pointer-to-member type. A handler of reference to array or function type is never a match for any exception object (7.6.18). — end note]

[Example 1:

```cpp
class Matherr { /* ... */ virtual void vf(); };  
class Overflow: public Matherr { /* ... */ };  
class Underflow: public Matherr { /* ... */ };  
class Zerodivide: public Matherr { /* ... */ };  

void f() {
  try {
    g();
  } catch (Overflow oo) {
    // ...
  } catch (Matherr mm) {
    // ...
  }
}
```

Here, the Overflow handler will catch exceptions of type Overflow and the Matherr handler will catch exceptions of type Matherr and of all types publicly derived from Matherr including exceptions of type Underflow and Zerodivide. — end example]

4 The handlers for a try block are tried in order of appearance.

[Note 2: This makes it possible to write handlers that can never be executed, for example by placing a handler for a final derived class after a handler for a corresponding unambiguous public base class. — end note]

5 A ... in a handler’s exception-declaration functions similarly to ... in a function parameter declaration; it specifies a match for any exception. If present, a ... handler shall be the last handler for its try block.

6 If no match is found among the handlers for a try block, the search for a matching handler continues in a dynamically surrounding try block of the same thread.

7 A handler is considered active when initialization is complete for the parameter (if any) of the catch clause.

[Note 3: The stack will have been unwound at that point. — end note]

Also, an implicit handler is considered active when the function std::terminate is entered due to a throw. A handler is no longer considered active when the catch clause exits.

8 The exception with the most recently activated handler that is still active is called the currently handled exception.

9 If no matching handler is found, the function std::terminate is called; whether or not the stack is unwound before this call to std::terminate is implementation-defined (14.6.2).

10 Referring to any non-static member or base class of an object in the handler for a function-try-block of a constructor or destructor for that object results in undefined behavior.

11 The scope and lifetime of the parameters of a function or constructor extend into the handlers of a function-try-block.

12 Exceptions thrown in destructors of objects with static storage duration or in constructors of namespace-scope objects with static storage duration are not caught by a function-try-block on the main function (6.9.3.1). Exceptions thrown in destructors of objects with thread storage duration or in constructors of namespace-scope objects with thread storage duration are not caught by a function-try-block on the initial function of the thread.

13 If a return statement (8.7.4) appears in a handler of the function-try-block of a constructor, the program is ill-formed.

14 The currently handled exception is rethrown if control reaches the end of a handler of the function-try-block of a constructor or destructor. Otherwise, flowing off the end of the compound-statement of a handler of a function-try-block is equivalent to flowing off the end of the compound-statement of that function (see 8.7.4).

15 The variable declared by the exception-declaration, of type cv T or cv T&, is initialized from the exception object, of type E, as follows:

(15.1) — if T is a base class of E, the variable is copy-initialized (9.4) from the corresponding base class subobject of the exception object;
— otherwise, the variable is copy-initialized (9.4) from the exception object.

The lifetime of the variable ends when the handler exits, after the destruction of any objects with automatic storage duration initialized within the handler.

When the handler declares an object, any changes to that object will not affect the exception object. When the handler declares a reference to an object, any changes to the referenced object are changes to the exception object and will have effect should that object be rethrown.

14.5 Exception specifications

The predicate indicating whether a function cannot exit via an exception is called the exception specification of the function. If the predicate is false, the function has a potentially-throwing exception specification, otherwise it has a non-throwing exception specification. The exception specification is either defined implicitly, or defined explicitly by using a noexcept-specifier as a suffix of a function declarator (9.3.4.6).

```
nonexcept-specifier:
   noexcept ( constant-expression )
   noexcept
```

In a noexcept-specifier, the constant-expression, if supplied, shall be a contextually converted constant expression of type bool (7.7); that constant expression is the exception specification of the function type in which the noexcept-specifier appears. A ( token that follows noexcept is part of the noexcept-specifier and does not commence an initializer (9.4). The noexcept-specifier noexcept without a constant-expression is equivalent to the noexcept-specifier noexcept(true).

If a declaration of a function does not have a noexcept-specifier, the declaration has a potentially throwing exception specification unless it is a destructor or a deallocation function or is defaulted on its first declaration, in which cases the exception specification is as specified below and no other declaration for that function shall have a noexcept-specifier. In an explicit instantiation (13.9.3) a noexcept-specifier may be specified, but is not required. If a noexcept-specifier is specified in an explicit instantiation directive, the exception specification shall be the same as the exception specification of all other declarations of that function. A diagnostic is required only if the exception specifications are not the same within a single translation unit.

If a virtual function has a non-throwing exception specification, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall have a non-throwing exception specification, unless the overriding function is defined as deleted.

```
[Example 1:]

struct B {
   virtual void f() noexcept;
   virtual void g();
   virtual void h() noexcept = delete;
};

struct D: B {
   void f(); // error
   void g() noexcept; // OK
   void h() = delete; // OK
};
```

The declaration of D::f is ill-formed because it has a potentially-throwing exception specification, whereas B::f has a non-throwing exception specification. — end example]

Whenever an exception is thrown and the search for a handler (14.4) encounters the outermost block of a function with a non-throwing exception specification, the function std::terminate is called (14.6.2).

[Note 1: An implementation is not permitted to reject an expression merely because, when executed, it throws or might throw an exception from a function with a non-throwing exception specification. — end note]

```
[Example 2:]

extern void f(); // potentially-throwing

void g() noexcept {
   f(); // valid, even if f throws
   throw 42; // valid, effectively a call to std::terminate
}
```

§ 14.5
An expression $E$ is potentially-throwing if

1. $E$ is a function call (7.6.1.3) whose postfix-expression has a function type, or a pointer-to-function type, with a potentially-throwing exception specification, or
2. $E$ implicitly invokes a function (such as an overloaded operator, an allocation function in a new-expression, a constructor for a function argument, or a destructor if $E$ is a full-expression (6.9.1)) that is potentially-throwing, or
3. $E$ is a throw-expression (7.6.18), or
4. $E$ is a dynamic_cast expression that casts to a reference type and requires a runtime check (7.6.1.7), or
5. $E$ is a typeid expression applied to a (possibly parenthesized) built-in unary $*$ operator applied to a pointer to a polymorphic class type (7.6.1.8), or
6. any of the immediate subexpressions (6.9.1) of $E$ is potentially-throwing.

An implicitly-declared constructor for a class $X$, or a constructor without a noexcept-specifier that is defaulted on its first declaration, has a potentially-throwing exception specification if and only if any of the following constructs is potentially-throwing:

1. a constructor selected by overload resolution in the implicit definition of the constructor for class $X$ to initialize a potentially constructed subobject, or
2. a subexpression of such an initialization, such as a default argument expression, or,
3. for a default constructor, a default member initializer.

[Note 2: Even though destructors for fully-constructed subobjects are invoked when an exception is thrown during the execution of a constructor (14.3), their exception specifications do not contribute to the exception specification of the constructor, because an exception thrown from such a destructor would call the function std::terminate rather than escape the constructor (14.2, 14.6.2). — end note]

The exception specification for an implicitly-declared destructor, or a destructor without a noexcept-specifier, is potentially-throwing if and only if any of the destructors for any of its potentially constructed subobjects is potentially-throwing or the destructor is virtual and the destructor of any virtual base class is potentially-throwing.

The exception specification for an implicitly-declared assignment operator, or an assignment-operator without a noexcept-specifier that is defaulted on its first declaration, is potentially-throwing if and only if the invocation of any assignment operator in the implicit definition is potentially-throwing.

A deallocation function (6.7.5.5.3) with no explicit noexcept-specifier has a non-throwing exception specification.

The exception specification for a comparison operator function (12.6.3) without a noexcept-specifier that is defaulted on its first declaration is potentially-throwing if and only if any expression in the implicit definition is potentially-throwing.

[Example 3:]

```cpp
class A {
    A(int = (A(5), 0)) noexcept;
    A(const A&) noexcept;
    A(A&&) noexcept;
    ~A();
};
class B {
    B() noexcept;
    B(const B&) = default; // implicit exception specification is noexcept(true)
    B(B&&, int = (throw 42, 0)) noexcept;
    ~B() noexcept(false);
};
int n = 7;
class D : public A, public B {
    int * p = new int[n];
    // D::D() potentially-throwing, as the new operator may throw bad_alloc or bad_array_new_length
    // D::D(const D&) non-throwing
    // D::D(D&&) potentially-throwing, as the default argument for B’s constructor may throw
};
```
Furthermore, if \( A::A() \) were virtual, the program would be ill-formed since a function that overrides a virtual function from a base class shall not have a potentially-throwing exception specification if the base class function has a non-throwing exception specification. — end example

An exception specification is considered to be needed when:

1. in an expression, the function is the unique lookup result or the selected member of a set of overloaded functions (6.5, 12.4, 12.5);
2. the function is odr-used (6.3) or, if it appears in an unevaluated operand, would be odr-used if the expression were potentially-evaluated;
3. the exception specification is compared to that of another declaration (e.g., an explicit specialization or an overriding virtual function);
4. the function is defined; or
5. the exception specification is needed for a defaulted function that calls the function.

[Note 3: A defaulted declaration does not require the exception specification of a base member function to be evaluated until the implicit exception specification of the derived function is needed, but an explicit noexcept-specifier needs the implicit exception specification to compare against. — end note]

The exception specification of a defaulted function is evaluated as described above only when needed; similarly, the noexcept-specifier of a specialization of a function template or member function of a class template is instantiated only when needed.

14.6 Special functions

14.6.1 General

The function `std::terminate` (14.6.2) is used by the exception handling mechanism for coping with errors related to the exception handling mechanism itself. The function `std::current_exception()` (17.9.7) and the class `std::nested_exception` (17.9.8) can be used by a program to capture the currently handled exception.

14.6.2 The `std::terminate` function

In some situations exception handling is abandoned for less subtle error handling techniques.

[Note 1: These situations are:

1. when the exception handling mechanism, after completing the initialization of the exception object but before activation of a handler for the exception (14.2), calls a function that exits via an exception, or
2. when the exception handling mechanism cannot find a handler for a thrown exception (14.4), or
3. when the search for a handler (14.4) encounters the outermost block of a function with a non-throwing exception specification (14.5), or
4. when the destruction of an object during stack unwinding (14.3) terminates by throwing an exception, or
5. when initialization of a non-local variable with static or thread storage duration (6.9.3.3) exits via an exception, or
6. when destruction of an object with static or thread storage duration exits via an exception (6.9.3.4), or
7. when execution of a function registered with `std::atexit` or `std::at_quick_exit` exits via an exception (17.5), or
8. when a throw-expression (7.6.18) with no operand attempts to rethrow an exception and no exception is being handled (14.2), or
9. when the function `std::nested_exception::rethrow_nested` is called for an object that has captured no exception (17.9.8), or
10. when execution of the initial function of a thread exits via an exception (32.4.3.3), or
11. for a parallel algorithm whose ExecutionPolicy specifies such behavior (20.18.4, 20.18.5, 20.18.6), when execution of an element access function (25.3.1) of the parallel algorithm exits via an exception (25.3.4), or
12. when the destructor or the move assignment operator is invoked on an object of type `std::thread` that refers to a joinable thread (32.4.3.4, 32.4.3.5), or]
— when a call to a `wait()`, `wait_until()`, or `wait_for()` function on a condition variable (32.6.4, 32.6.5) fails to meet a postcondition.

— end note

In such cases, the function `std::terminate` is called (17.9.5). In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before `std::terminate` is called. In the situation where the search for a handler (14.4) encounters the outermost block of a function with a non-throwing exception specification (14.5), it is implementation-defined whether the stack is unwound, unwound partially, or not unwound at all before the function `std::terminate` is called. In all other situations, the stack shall not be unwound before the function `std::terminate` is called. An implementation is not permitted to finish stack unwinding prematurely based on a determination that the unwind process will eventually cause a call to the function `std::terminate`.

### 14.6.3 The `std::uncaught_exceptions` function

[except.uncaught]

An exception is considered uncaught after completing the initialization of the exception object (14.2) until completing the activation of a handler for the exception (14.4).

[Note 1: As a consequence, an exception is considered uncaught during any stack unwinding resulting from it being thrown. — end note]

If an exception is rethrown (7.6.18, 17.9.7), it is considered uncaught from the point of rethrow until the rethrown exception is caught. The function `std::uncaught_exceptions` (17.9.6) returns the number of uncaught exceptions in the current thread.
15 Preprocessing directives

15.1 Preamble

preprocessing-file:
  group_opt
  module-file

module-file:
  pp-global-module-fragment_opt pp-module group_opt pp-private-module-fragment_opt

pp-global-module-fragment:
  module ; new-line group_opt

pp-private-module-fragment:
  module : private ; new-line group_opt

group:
  group-part
  group group-part

group-part:
  control-line
  if-section
  text-line
  # conditionally-supported-directive

control-line:
  # include pp-tokens new-line
  # import
  # define identifier replacement-list new-line
  # define identifier lparen identifier-list_opt ) replacement-list new-line
  # define identifier lparen ... ) replacement-list new-line
  # define identifier lparen identifier-list , ... ) replacement-list new-line
  # undef identifier new-line
  # line pp-tokens new-line
  # error pp-tokens_opt new-line
  # pragma pp-tokens_opt new-line
  # new-line

if-section:
  if-group elif-groups_opt else-group_opt endif-line

if-group:
  # if constant-expression new-line group_opt
  # ifdef identifier new-line group_opt
  # ifndef identifier new-line group_opt

elif-groups:
  elif-group
  elif-groups elif-group

elif-group:
  # elif constant-expression new-line group_opt

else-group:
  # else new-line group_opt

endif-line:
  # endif new-line

text-line:
  pp-tokens_opt new-line

conditionally-supported-directive:
  pp-tokens new-line

lparen:
  a ( character not immediately preceded by whitespace
A preprocessing directive consists of a sequence of preprocessing tokens that satisfies the following constraints:

At the start of translation phase 4, the first token in the sequence, referred to as a directive-introducing token, begins with the first character in the source file (optionally after whitespace containing no new-line characters) or follows whitespace containing at least one new-line character, and is

1. a `#` preprocessing token, or
2. an `import` preprocessing token immediately followed on the same logical line by a `header-name`, `<`, `identifier`, `string-literal`, or `:` preprocessing token, or
3. an `module` preprocessing token immediately followed on the same logical line by an `identifier`, `;`, or `;` preprocessing token, or
4. an `export` preprocessing token immediately followed on the same logical line by one of the two preceding forms.

The last token in the sequence is the first token within the sequence that is immediately followed by whitespace containing a new-line character.\[1\]

[Note 1: A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro. — end note]

**Example 1:**

```plaintext
#include // preprocessing directive
module ; // preprocessing directive
export module leftright; // preprocessing directive
import <string> ; // preprocessing directive
export import "squee"; // preprocessing directive
import rightpad; // preprocessing directive
import :part; // preprocessing directive
module // not a preprocessing directive
; // not a preprocessing directive
export // not a preprocessing directive
import // not a preprocessing directive
foo; // not a preprocessing directive
export // not a preprocessing directive
import foo; // not a preprocessing directive (ill-formed at phase 7)
import :: // not a preprocessing directive
import -> // not a preprocessing directive

— end example]
```

2. A sequence of preprocessing tokens is only a text-line if it does not begin with a directive-introducing token. A sequence of preprocessing tokens is only a conditionally-supported-directive if it does not begin with any of the directive names appearing after a `#` in the syntax. A conditionally-supported-directive is conditionally-supported with implementation-defined semantics.

3. At the start of phase 4 of translation, the group of a pp-global-module-fragment shall contain neither a text-line nor a pp-import.

---

146) Thus, preprocessing directives are commonly called “lines”. These “lines” have no other syntactic significance, as all whitespace is equivalent except in certain situations during preprocessing (see the `#` character string literal creation operator in 15.6.3, for example).
When in a group that is skipped (15.2), the directive syntax is relaxed to allow any sequence of preprocessing
tokens to occur between the directive name and the following new-line character.

The only whitespace characters that shall appear between preprocessing tokens within a preprocessing
directive (from just after the directive-introducing token through just before the terminating new-line
character) are space and horizontal-tab (including spaces that have replaced comments or possibly other
whitespace characters in translation phase 3).

The implementation can process and skip sections of source files conditionally, include other source files,
import macros from header units, and replace macros. These capabilities are called preprocessing, because
conceptually they occur before translation of the resulting translation unit.

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise
stated.

[Example 2: In:

```c
#define EMPTY
EMPTY # include <file.h>
```

the sequence of preprocessing tokens on the second line is not a preprocessing directive, because it does not begin
with a # at the start of translation phase 4, even though it will do so after the macro `EMPTY` has been replaced. — end
example]

15.2 Conditional inclusion

```c
defined-macro-expression:
    defined identifier
    defined ( identifier )

h-preprocessing-token:
    any preprocessing-token other than >

h-pp-tokens:
    h-preprocessing-token
    h-pp-tokens h-preprocessing-token

header-name-tokens:
    string-literal
    < h-pp-tokens >

has-include-expression:
    __has_include ( header-name )
    __has_include ( header-name-tokens )

has-attribute-expression:
    __has_cpp_attribute ( pp-tokens )
```

1 The expression that controls conditional inclusion shall be an integral constant expression except that
identifiers (including those lexically identical to keywords) are interpreted as described below and it may
contain zero or more defined-macro-expressions and/or has-include-expressions and/or has-attribute-expressions
as unary operator expressions.

2 A defined-macro-expression evaluates to 1 if the identifier is currently defined as a macro name (that is, if it
is predefined or if it has one or more active macro definitions (15.5), for example because it has been the
subject of a `#define` preprocessing directive without an intervening `#undef` directive with the same subject
identifier), 0 if it is not.

3 The second form of has-include-expression is considered only if the first form does not match, in which case
the preprocessing tokens are processed just as in normal text.

4 The header or source file identified by the parenthesized preprocessing token sequence in each contained
has-include-expression is searched for as if that preprocessing token sequence were the pp-tokens in a `#include`
directive, except that no further macro expansion is performed. If such a directive would not satisfy the
syntactic requirements of a `#include` directive, the program is ill-formed. The has-include-expression evaluates
to 1 if the search for the source file succeeds, and to 0 if the search fails.

5 Each has-attribute-expression is replaced by a non-zero pp-number matching the form of an integer-literal if
the implementation supports an attribute with the name specified by interpreting the pp-tokens, after macro

---

147) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not
macro names — there simply are no keywords, enumeration constants, etc.
expansion, as an attribute-token, and by 0 otherwise. The program is ill-formed if the pp-tokens do not match the form of an attribute-token.

6 For an attribute specified in this document, the value of the has-attribute-expression is given by Table 18. For other attributes recognized by the implementation, the value is implementation-defined.

[Note 1: It is expected that the availability of an attribute can be detected by any non-zero result. — end note]

Table 18: __has_cpp_attribute values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>carries_dependency</td>
<td>200809L</td>
</tr>
<tr>
<td>deprecated</td>
<td>201309L</td>
</tr>
<tr>
<td>fallthrough</td>
<td>201603L</td>
</tr>
<tr>
<td>likely</td>
<td>201803L</td>
</tr>
<tr>
<td>maybe_unused</td>
<td>201603L</td>
</tr>
<tr>
<td>no_unique_address</td>
<td>201803L</td>
</tr>
<tr>
<td>nodiscard</td>
<td>201907L</td>
</tr>
<tr>
<td>noreturn</td>
<td>200809L</td>
</tr>
<tr>
<td>unlikely</td>
<td>201803L</td>
</tr>
</tbody>
</table>

7 The #ifdef and #ifndef directives, and the defined conditional inclusion operator, shall treat __has_-include and __has_cpp_attribute as if they were the names of defined macros. The identifiers __has_-include and __has_cpp_attribute shall not appear in any context not mentioned in this subclause.

8 Each preprocessing token that remains (in the list of preprocessing tokens that will become the controlling expression) after all macro replacements have occurred shall be in the lexical form of a token (5.6).

9 Preprocessing directives of the forms

```c
#define constant-expression new-line group
```

check whether the controlling constant expression evaluates to nonzero.

10 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the defined unary operator), just as in normal text. If the token defined is generated as a result of this replacement process or use of the defined unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined.

11 After all replacements due to macro expansion and evaluations of defined-macro-expressions, has-include-expressions, and has-attribute-expressions have been performed, all remaining identifiers and keywords, except for true and false, are replaced with the pp-number 0, and then each preprocessing token is converted into a token.

[Note 2: An alternative token (5.5) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not subject to this replacement. — end note]

12 The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 7.7 using arithmetic that has at least the ranges specified in 17.3. For the purposes of this token conversion and evaluation all signed and unsigned integer types act as if they have the same representation as, respectively, intmax_t or uintmax_t (17.4).

[Note 3: Thus on an implementation where std::numeric_limits<int>::max() is 0x7FFF and std::numeric_limits<unsigned int>::max() is 0xFFFF, the integer literal 0x8000 is signed and positive within a #if expression even though it is unsigned in translation phase 7 (5.2). — end note]

This includes interpreting character-literals, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character-literals matches the value obtained when an identical character-literal occurs in an expression (other than within a #if or #elif directive) is implementation-defined.

[Note 4: Thus, the constant expression in the following #if directive and if statement (8.5.2) is not guaranteed to evaluate to the same value in these two contexts:

```c
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

§ 15.2
Also, whether a single-character character-literal may have a negative value is implementation-defined. Each subexpression with type bool is subjected to integral promotion before processing continues.

Preprocessing directives of the forms

```
#define identifier new-line group_opt
#else new-line group_opt
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to #if defined identifier and #if !defined identifier respectively.

Each directive’s condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives’ preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed; any following groups are skipped and their controlling directives are processed as if they were in a group that is skipped. If none of the conditions evaluates to true, and there is a #else directive, the group controlled by the #else is processed; lacking a #else directive, all the groups until the #endif are skipped.148

**Example 1:** This demonstrates a way to include a library optional facility only if it is available:

```
#include <optional>
#if __cpp_lib_optional >= 201603
#define have_optional 1
#endif
#elif __has_include(<experimental/optional>)
#include <optional>
#define have_optional 1
#define experimental_optional 1
#elif __cpp_lib_experimental_optional >= 201411
#endif
#ifndef have_optional
#define have_optional 0
#endif
```

**Example 2:** This demonstrates a way to use the attribute [[acme::deprecated]] only if it is available.

```
#define ATTR_DEPRECATED(msg) [[acme::deprecated(msg)]]
```

```
ATTR_DEPRECATED("This function is deprecated") void anvil();
```

---

15.3 **Source file inclusion**

A #include directive shall identify a header or source file that can be processed by the implementation.

A preprocessing directive of the form

```
#include <h-char-sequence> new-line
```

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

A preprocessing directive of the form

```
#include "q-char-sequence" new-line
```

---

148 As indicated by the syntax, a preprocessing token cannot follow a #else or #endif directive before the terminating new-line character. However, comments can appear anywhere in a source file, including within a preprocessing directive.
causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

```
#include <h-char-sequence> new-line
```

with the identical contained sequence (including > characters, if any) from the original directive.

4 A preprocessing directive of the form

```
#include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (i.e., each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined.\(^{149}\) The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single header name preprocessing token is implementation-defined.

5 The implementation shall provide unique mappings for sequences consisting of one or more nondigits or digits (5.10) followed by a period (.) and a single nondigit. The first character shall not be a digit. The implementation may ignore distinctions of alphabetical case.

6 A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit.

7 If the header identified by the `header-name` denotes an importable header (10.3), it is implementation-defined whether the `#include` preprocessing directive is instead replaced by an `import` directive (15.5) of the form

```
import header-name ; new-line
```

8 [Note 1: An implementation can provide a mechanism for making arbitrary source files available to the < > search. However, using the < > form for headers provided with the implementation and the " " form for sources outside the control of the implementation achieves wider portability. For instance:

```
#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"
```

—end note]

9 [Example 1: This illustrates macro-replaced `#include` directives:

```
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h" // and so on
#else
#define INCFILE "versN.h"
#endif
#include INCFILE
```

—end example]
15.5 Header unit importation

pp-import:
  export_opt import header-name pp-tokens_opt ; new-line
  export_opt import header-name-tokens pp-tokens_opt ; new-line
  export_opt import pp-tokens ; new-line

1 A pp-import shall not appear in a context where import or (if it is the first token of the pp-import) export is an identifier defined as an object-like macro.

2 The preprocessing tokens after the import preprocessing token in the import control-line are processed just as in normal text (i.e., each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). An import directive matching the first two forms of a pp-import instructs the preprocessor to import macros from the header unit (10.3) denoted by the header-name. The point of macro import for the first two forms of pp-import is immediately after the new-line terminating the pp-import. The last form of pp-import is only considered if the first two forms did not match.

3 If a pp-import is produced by source file inclusion (including by the rewrite produced when a #include directive names an importable header) while processing the group of a module-file, the program is ill-formed.

4 In all three forms of pp-import, the import and export (if it exists) preprocessing tokens are replaced by the import-keyword and export-keyword preprocessing tokens respectively.

5 Each define directive encountered when preprocessing each translation unit in a program results in a distinct macro definition.

6 A macro directive is active at a source location if it has a point of definition in that translation unit preceding the location, and does not have a point of undefinition in that translation unit preceding the location.

7 If a macro would be replaced or redefined, and multiple macro definitions are active for that macro name, the active macro definitions shall all be valid redefinitions of the same macro (15.6).

8 [Example 1:

Importable header "a.h":

#define X 123 // #1
#define Y 45 // #2
#define Z a // #3
#undef X // point of undefined of #1 in "a.h"
15.6 Macro replacement

15.6.1 General

Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and whitespace separation, where all whitespace separations are considered identical.

An identifier currently defined as an object-like macro (see below) may be redefined by another #define preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical, otherwise the program is ill-formed. Likewise, an identifier currently defined as a function-like macro (see below) may be redefined by another #define preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical, otherwise the program is ill-formed.

Example 1: The following sequence is valid:

```c
#define OBJ_LIKE (1-1)
#define OBJ_LIKE /* whitespace */ (1-1) /* other */
#define FUNC_LIKE(a) ( a )
#define FUNC_LIKE( a )/* note the whitespace */ \ 
a /* other stuff on this line */
```

But the following redefinitions are invalid:

```c
#define OBJ_LIKE (0) // different token sequence
#define OBJ_LIKE (1 - 1) // different whitespace
#define FUNC_LIKE(b) ( a ) // different parameter usage
#define FUNC_LIKE(b) ( b ) // different parameter spelling
```

There shall be whitespace between the identifier and the replacement list in the definition of an object-like macro.

If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be at least as many arguments in the invocation as there are parameters in the macro definition (excluding the ...). There shall exist a ) preprocessing token that terminates the invocation.

The identifiers __VA_ARGS__ and __VA_OPT__ shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

A parameter identifier in a function-like macro shall be uniquely declared within its scope.

The identifier immediately following the define is called the macro name. There is one name space for macro names. Any whitespace characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

A preprocessing directive of the form
defines an object-like macro that causes each subsequent instance of the macro name\textsuperscript{150} to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.\textsuperscript{151} The replacement list is then rescanned for more macro names as specified below.

\textbf{Example 2}: The simplest use of this facility is to define a “manifest constant”, as in

\begin{verbatim}
#define TABSIZE 100
int table[TABSIZE];
\end{verbatim}

A preprocessing directive of the form

\begin{verbatim}
#define identifier lparen identifier-list opt replacement-list new-line
#define identifier lparen ... ) replacement-list new-line
#define identifier lparen identifier-list , ... ) replacement-list new-line
\end{verbatim}

defines a function-like macro with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the \texttt{#define} preprocessing directive. Each subsequent instance of the function-like macro name followed by a \texttt{(} as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching \texttt{)} preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal whitespace character.

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives,\textsuperscript{152} the behavior is undefined.

\textbf{Example 3}: The following defines a function-like macro whose value is the maximum of its arguments. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

\begin{verbatim}
#define max(a, b) ((a) > (b) ? (a) : (b))
\end{verbatim}

The parentheses ensure that the arguments and the resulting expression are bound properly. —end example]

If there is a \texttt{...} immediately preceding the \texttt{)} in the function-like macro definition, then the trailing arguments (if any), including any separating comma preprocessing tokens, are merged to form a single item: the \textit{variable arguments}. The number of arguments so combined is such that, following merger, the number of arguments is either equal to or one more than the number of parameters in the macro definition (excluding the \texttt{...}).

\section{Argument substitution} [cpp.subst]

\texttt{va-opt-replacement}:

\begin{verbatim}
__VA_OPT__ ( pp-tokensopt )
\end{verbatim}

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. For each parameter in the replacement list that is neither preceded by a \texttt{#} or \texttt{##} preprocessing token nor followed by a \texttt{##} preprocessing token, the preprocessing tokens naming the parameter are replaced by a token sequence determined as follows:

\begin{enumerate}
\item If the parameter is of the form \texttt{va-opt-replacement}, the replacement preprocessing tokens are the preprocessing token sequence for the corresponding argument, as specified below.
\item Otherwise, the replacement preprocessing tokens are the preprocessing tokens of corresponding argument after all macros contained therein have been expanded. The argument’s preprocessing tokens are completely macro replaced before being substituted as if they formed the rest of the preprocessing file with no other preprocessing tokens being available.
\end{enumerate}

\textsuperscript{150}Since, by macro-replacement time, all \textit{character-} and \textit{string-} literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 5.2, translation phases), they are never scanned for macro names or parameters.

\textsuperscript{151}An alternative token (5.5) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not possible to define a macro whose name is the same as that of an alternative token.

\textsuperscript{152}A \texttt{conditionally-supported-directive} is a preprocessing directive regardless of whether the implementation supports it.
Example 1:

```c
#define LPAREN() (
#define G(Q) 42
#define F(R, X, ...) __VA_OPT__(G R X) )

int x = F(LPAREN(), 0, 0:-);  // replaced by int x = 42;
```

An identifier `__VA_ARGS__` that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

Example 2:

```c
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test) ? puts(#test) : printf(__VA_ARGS__))

debug("Flag");
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```c
fprintf(stderr, "Flag");
putc(stderr, "X = %d\n", x);
puts("The first, second, and third items."");
((x>y) ? puts("x>y") : printf("x is %d but y is %d", x, y));
```

Example 3:

```c
#define F(...) f(0 __VA_OPT__(,) __VA_ARGS__)
#define G(X, ...) f(0, X __VA_OPT__(,) __VA_ARGS__)
#define SDEF(sname, ...) S sname __VA_OPT__(= { __VA_ARGS__ })
#define EMP
F(a, b, c)  // replaced by f(0, a, b, c)
F()  // replaced by f(0)
F(EMP)  // replaced by f(0)
G(a, b, c)  // replaced by f(0, a, b, c)
G(a)  // replaced by f(0, a)
G(a)  // replaced by f(0, a)
SDEF(foo);  // replaced by S foo;
SDEF(bar, 1, 2);  // replaced by S bar = { 1, 2 };
```

The identifier `__VA_OPT__` shall always occur as part of the preprocessing token sequence `va-opt-replacement`; its closing ) is determined by skipping intervening pairs of matching left and right parentheses in its pp-tokens. The pp-tokens of a va-opt-replacement shall not contain `__VA_OPT__`. If the pp-tokens would be ill-formed as the replacement list of the current function-like macro, the program is ill-formed. A va-opt-replacement is treated as if it were a parameter, and the preprocessing token sequence for the corresponding argument is defined as follows. If the substitution of `__VA_ARGS__` as neither an operand of `#` nor `##` consists of no preprocessing tokens, the argument consists of a single placemarker preprocessing token (15.6.4, 15.6.5). Otherwise, the argument consists of the results of the expansion of the contained pp-tokens as the replacement list of the current function-like macro before removal of placemarker tokens, rescanning, and further replacement.

Note 1: The placemarker tokens are removed before stringization (15.6.3), and can be removed by rescanning and further replacement (15.6.5). — end note

Example 3:

```c
#define H1(X, ...) X __VA_OPT__(##) __VA_ARGS__  // error: ## may not appear at the beginning of a replacement list (15.6.4)
#define H2(X, Y, ...) __VA_OPT__(X ## Y,) __VA_ARGS__
H2(a, b, c, d)  // replaced by ab, c, d
```
Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

A character string literal is a string-literal with no prefix. If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument (excluding placemarker tokens). Let the stringizing argument be the preprocessing token sequence for the corresponding argument with placemarker tokens removed. Each occurrence of whitespace between the stringizing argument’s preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token comprising the stringizing argument is deleted. Otherwise, the original spelling of each preprocessing token in the stringizing argument is retained in the character string literal, except for special handling for producing the spelling of string-literals and character-literals: a \ character is inserted before each " and \ character of a character-literal or string-literal (including the delimiting " characters). If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty stringizing argument is "".

The order of evaluation of # and ## operators is unspecified.

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemakers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

15.6.4 The ## operator

1 A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

2 If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument’s preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead.\(^{153}\)

3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemakers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

\(^{153}\) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.
debug(1, 2);
fprintf(stderr, "%d, %d\n", x1, x2);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW);

results in

printf("x\n1\n= %d,\nx\n2\n= %s", x1, x2);
#include "vers2.h" (after macro replacement, before file access)
"hello\n;\n"hello\n, world"
or, after concatenation of the character string literals,

printf("x1= %d, x2= %s", x1, x2);
#include "vers2.h" (after macro replacement, before file access)
"hello\n;\n"hello, world"

Space around the # and ## tokens in the macro definition is optional. — end example] 5

Example 2: In the following fragment:

#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
char p[ ] = join(x, y);
// equivalent to char p[ ] = "x ## y"

The expansion produces, at various stages:

join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"

In other words, expanding hash_hash produces a new token, consisting of two adjacent sharp signs, but this new
token is not the ## operator. — end example] 6

Example 3: To illustrate the rules for placemarker preprocessing tokens, the sequence

#define t(x,y,z) x ## y ## z
int j[ ] = { t(1,2,3), t(4,5), t(6,7), t(8,9),
t(10,,), t(,,11), t(,,12), t(,,) };

results in

int j[ ] = { 123, 45, 67, 89,
10, 11, 12, };
— end example]

15.6.5 Rescanning and further replacement [cpp.rescan]

After all parameters in the replacement list have been substituted and # and ## processing has taken place, all
placemarker preprocessing tokens are removed. Then the resulting preprocessing token sequence is rescanned,
along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

Example 1: The sequence

#define x 3
#define f(a) f(x * (a))
#undef x
#define x 2
#define g f
#define z [0]
#define h g(~
#define m(a) a(w)
#define w 0,1
#define t(a) a

§ 15.6.5 472
#define p() int
#define q(x) x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(x)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
(f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };  
char c[2][6] = { str(hello), str() };
results in
f(2 * (y+1)) + f(2 * (f(2 * (z[0]))) % f(2 * (0)) + t(1));
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5, };  
char c[2][6] = { "hello", "" };

— end example

3 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest
of the source file’s preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter
the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing
tokens are no longer available for further replacement even if they are later (re)examined in contexts in which
that macro name preprocessing token would otherwise have been replaced.

4 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing
directive even if it resembles one, but all pragma unary operator expressions within it are then processed as
specified in 15.12 below.

15.6.6 Scope of macro definitions [cpp.scope]

1 A macro definition lasts (independent of block structure) until a corresponding #undef directive is encountered
or (if none is encountered) until the end of the translation unit. Macro definitions have no significance after
translation phase 4.

2 A preprocessing directive of the form

    # undef identifier new-line

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier
is not currently defined as a macro name.

15.7 Line control [cpp.line]

1 The string-literal of a #line directive, if present, shall be a character string literal.

2 The line number of the current source line is one greater than the number of new-line characters read or
introduced in translation phase 1 (5.2) while processing the source file to the current token.

3 A preprocessing directive of the form

    # line digit-sequence new-line

causes the implementation to behave as if the following sequence of source lines begins with a source line that
has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence
specifies zero or a number greater than 2147483647, the behavior is undefined.

4 A preprocessing directive of the form

    # line digit-sequence " s-char-sequence opt " new-line

sets the presumed line number similarly and changes the presumed name of the source file to be the contents
of the character string literal.

5 A preprocessing directive of the form

    # line pp-tokens new-line

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after line on the
directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by
its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match
one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.
15.8 Error directive

A preprocessing directive of the form

```cpp
#error pp-tokens new-line
```

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens, and renders the program ill-formed.

15.9 Pragma directive

A preprocessing directive of the form

```cpp
#pragma pp-tokens new-line
```

causes the implementation to behave in an implementation-defined manner. The behavior may cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any pragma that is not recognized by the implementation is ignored.

15.10 Null directive

A preprocessing directive of the form

```cpp
# new-line
```

has no effect.

15.11 Predefined macro names

The following macro names shall be defined by the implementation:

```cpp
__cplusplus
```

The integer literal `202002L`.

[Note 1: Future revisions of C++ will replace the value of this macro with a greater value. — end note]

The names listed in Table 19.

The macros defined in Table 19 shall be defined to the corresponding integer literal.

[Note 2: Future revisions of C++ might replace the values of these macros with greater values. — end note]

```cpp
__DATE__
```

The date of translation of the source file: a character string literal of the form "Mmm dd yyyy", where the names of the months are the same as those generated by the `asctime` function, and the first character of `dd` is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

```cpp
__FILE__
```

The presumed name of the current source file (a character string literal).

154

```cpp
__LINE__
```

The presumed line number (within the current source file) of the current source line (an integer literal).

155

```cpp
__STDC_HOSTED__
```

The integer literal 1 if the implementation is a hosted implementation or the integer literal 0 if it is a freestanding implementation (4.1).

```cpp
__STDCPP_DEFAULT_NEW_ALIGNMENT__
```

An integer literal of type `std::size_t` whose value is the alignment guaranteed by a call to `operator new(std::size_t)` or `operator new[](std::size_t)`.

[Note 3: Larger alignments will be passed to `operator new(std::size_t, std::align_val_t)` etc. (7.6.2.8). — end note]

```cpp
__TIME__
```

The time of translation of the source file: a character string literal of the form "hh:mm:ss" as in the time generated by the `asctime` function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

154) The presumed source file name can be changed by the `#line` directive.

155) The presumed line number can be changed by the `#line` directive.
<table>
<thead>
<tr>
<th>Macro name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_aggregate_bases</td>
<td>201603L</td>
</tr>
<tr>
<td>__cpp_aggregate_nsdmi</td>
<td>201304L</td>
</tr>
<tr>
<td>__cpp_aggregate_paren_init</td>
<td>201902L</td>
</tr>
<tr>
<td>__cpp_alias_templates</td>
<td>200704L</td>
</tr>
<tr>
<td>__cpp_aligned_new</td>
<td>201606L</td>
</tr>
<tr>
<td>__cpp Attributes</td>
<td>200809L</td>
</tr>
<tr>
<td>__cpp_binary_literals</td>
<td>201304L</td>
</tr>
<tr>
<td>__cpp_capture_star_this</td>
<td>201603L</td>
</tr>
<tr>
<td>__cpp_char8_t</td>
<td>201811L</td>
</tr>
<tr>
<td>__cpp_concepts</td>
<td>201907L</td>
</tr>
<tr>
<td>__cpp_conditional_explicit</td>
<td>201806L</td>
</tr>
<tr>
<td>__cpp_constexpr</td>
<td>201907L</td>
</tr>
<tr>
<td>__cpp_constexpr_dynamic_alloc</td>
<td>201907L</td>
</tr>
<tr>
<td>__cpp_constexpr_in_decltype</td>
<td>201711L</td>
</tr>
<tr>
<td>__cpp_consteval</td>
<td>201811L</td>
</tr>
<tr>
<td>__cpp_constinit</td>
<td>201907L</td>
</tr>
<tr>
<td>__cpp_decltype</td>
<td>200707L</td>
</tr>
<tr>
<td>__cpp_decltype_auto</td>
<td>201304L</td>
</tr>
<tr>
<td>__cpp_deduction_guides</td>
<td>201907L</td>
</tr>
<tr>
<td>__cpp_delegating_constructors</td>
<td>200604L</td>
</tr>
<tr>
<td>__cpp_designated_initializers</td>
<td>201707L</td>
</tr>
<tr>
<td>__cpp_enumerator_attributes</td>
<td>201411L</td>
</tr>
<tr>
<td>__cpp_fold_expressions</td>
<td>201603L</td>
</tr>
<tr>
<td>__cpp_generic_lambdas</td>
<td>201707L</td>
</tr>
<tr>
<td>__cpp_guaranteed_copy_elision</td>
<td>201606L</td>
</tr>
<tr>
<td>__cpp_hex_float</td>
<td>201603L</td>
</tr>
<tr>
<td>__cpp_if_constexpr</td>
<td>201606L</td>
</tr>
<tr>
<td>__cpp_impl_coroutine</td>
<td>201902L</td>
</tr>
<tr>
<td>__cpp_impl_destroying_delete</td>
<td>201806L</td>
</tr>
<tr>
<td>__cpp_impl_three_way_comparison</td>
<td>201907L</td>
</tr>
<tr>
<td>__cpp_inheriting_constructors</td>
<td>201511L</td>
</tr>
<tr>
<td>__cpp_init_captures</td>
<td>201803L</td>
</tr>
<tr>
<td>__cpp_initializer_lists</td>
<td>200806L</td>
</tr>
<tr>
<td>__cpp_inline_variables</td>
<td>201606L</td>
</tr>
<tr>
<td>__cpp_lambdas</td>
<td>200907L</td>
</tr>
<tr>
<td>__cpp_modules</td>
<td>201907L</td>
</tr>
<tr>
<td>__cpp_namespace_attributes</td>
<td>201411L</td>
</tr>
<tr>
<td>__cpp_noexcept_function_type</td>
<td>201510L</td>
</tr>
<tr>
<td>__cpp_nontype_template_args</td>
<td>201911L</td>
</tr>
<tr>
<td>__cpp_nontype_template_parameter_auto</td>
<td>201606L</td>
</tr>
<tr>
<td>__cpp_nsdmi</td>
<td>200809L</td>
</tr>
<tr>
<td>__cpp_range_based_for</td>
<td>201603L</td>
</tr>
<tr>
<td>__cpp_raw_strings</td>
<td>200701L</td>
</tr>
<tr>
<td>__cpp_ref_qualifiers</td>
<td>200701L</td>
</tr>
<tr>
<td>__cpp_return_type_deduction</td>
<td>201304L</td>
</tr>
<tr>
<td>__cpp_rvalue_references</td>
<td>200610L</td>
</tr>
<tr>
<td>__cpp_sized_deallocation</td>
<td>201309L</td>
</tr>
<tr>
<td>__cpp_static_assert</td>
<td>201411L</td>
</tr>
<tr>
<td>__cpp_structured_deallocation</td>
<td>201606L</td>
</tr>
<tr>
<td>__cpp_template_template_args</td>
<td>201611L</td>
</tr>
<tr>
<td>__cpp_threadsafe_static_init</td>
<td>200806L</td>
</tr>
<tr>
<td>__cpp_unicode_characters</td>
<td>200704L</td>
</tr>
<tr>
<td>__cpp_unicode_literals</td>
<td>200710L</td>
</tr>
<tr>
<td>__cpp_user_defined_literals</td>
<td>200809L</td>
</tr>
</tbody>
</table>

§ 15.11
Table 19: Feature-test macros (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_using_enum</td>
<td>201907L</td>
</tr>
<tr>
<td>__cpp_variable_templates</td>
<td>201304L</td>
</tr>
<tr>
<td>__cpp_variadic_templates</td>
<td>200704L</td>
</tr>
<tr>
<td>__cpp_variadic_using</td>
<td>201611L</td>
</tr>
</tbody>
</table>

2 The following macro names are conditionally defined by the implementation:

__STDC__
Whether __STDC__ is predefined and if so, what its value is, are implementation-defined.

__STDC_MB_MIGHT_NEQ_WC__
The integer literal 1, intended to indicate that, in the encoding for wchar_t, a member of the basic character set need not have a code value equal to its value when used as the lone character in an ordinary character literal.

__STDC_VERSION__
Whether __STDC_VERSION__ is predefined and if so, what its value is, are implementation-defined.

__STDC_ISO_10646__
An integer literal of the form yyyymmL (for example, 199712L). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type wchar_t, has the same value as the code point of that character. The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month.

__STDCPP_STRICT_POINTER_SAFETY__
Defined, and has the value integer literal 1, if and only if the implementation has strict pointer safety (6.7.5.5.4).

__STDCPP_THREADS__
Defined, and has the value integer literal 1, if and only if a program can have more than one thread of execution (6.9.2).

3 The values of the predefined macros (except for __FILE__ and __LINE__) remain constant throughout the translation unit.

4 If any of the predefined macro names in this subclause, or the identifier defined, is the subject of a #define or a #undef preprocessing directive, the behavior is undefined. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

15.12 Pragma operator

A unary operator expression of the form:

Pragma ( string-literal )

is processed as follows: The string-literal is destringized by deleting the L prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence " by a double-quote, and replacing each escape sequence \\ by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the pp-tokens in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

[cppPragmaOp]

Example 1:

#pragma listing on ".\listing.dir"

can also be expressed as:

Pragma ( "listing on "..\listing.dir" )

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)
LISTING( ../listing.dir )
— end example]
16 Library introduction

16.1 General

This Clause describes the contents of the C++ standard library, how a well-formed C++ program makes use of the library, and how a conforming implementation may provide the entities in the library.

The following subclauses describe the method of description (16.3) and organization (16.4.2) of the library. 16.4, Clause 17 through Clause 32, and Annex D specify the contents of the library, as well as library requirements and constraints on both well-formed C++ programs and conforming implementations.

Detailed specifications for each of the components in the library are in Clause 17–Clause 32, as shown in Table 20.

Table 20: Library categories

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clause 17</td>
<td>Language support library</td>
</tr>
<tr>
<td>Clause 18</td>
<td>Concepts library</td>
</tr>
<tr>
<td>Clause 19</td>
<td>Diagnostics library</td>
</tr>
<tr>
<td>Clause 20</td>
<td>General utilities library</td>
</tr>
<tr>
<td>Clause 21</td>
<td>Strings library</td>
</tr>
<tr>
<td>Clause 22</td>
<td>Containers library</td>
</tr>
<tr>
<td>Clause 23</td>
<td>Iterators library</td>
</tr>
<tr>
<td>Clause 24</td>
<td>Ranges library</td>
</tr>
<tr>
<td>Clause 25</td>
<td>Algorithms library</td>
</tr>
<tr>
<td>Clause 26</td>
<td>Numerics library</td>
</tr>
<tr>
<td>Clause 27</td>
<td>Time library</td>
</tr>
<tr>
<td>Clause 28</td>
<td>Localization library</td>
</tr>
<tr>
<td>Clause 29</td>
<td>Input/output library</td>
</tr>
<tr>
<td>Clause 30</td>
<td>Regular expressions library</td>
</tr>
<tr>
<td>Clause 31</td>
<td>Atomic operations library</td>
</tr>
<tr>
<td>Clause 32</td>
<td>Thread support library</td>
</tr>
</tbody>
</table>

The language support library (Clause 17) provides components that are required by certain parts of the C++ language, such as memory allocation (7.6.2.8, 7.6.2.9) and exception processing (Clause 14).

The concepts library (Clause 18) describes library components that C++ programs may use to perform compile-time validation of template arguments and perform function dispatch based on properties of types.

The diagnostics library (Clause 19) provides a consistent framework for reporting errors in a C++ program, including predefined exception classes.

The general utilities library (Clause 20) includes components used by other library elements, such as a predefined storage allocator for dynamic storage management (6.7.5.5), and components used as infrastructure in C++ programs, such as tuples, function wrappers, and time facilities.

The strings library (Clause 21) provides support for manipulating text represented as sequences of type char, sequences of type char8_t, sequences of type char16_t, sequences of type char32_t, sequences of type wchar_t, and sequences of any other character-like type.

The localization library (Clause 28) provides extended internationalization support for text processing.

The containers (Clause 22), iterators (Clause 23), ranges (Clause 24), and algorithms (Clause 25) libraries provide a C++ program with access to a subset of the most widely used algorithms and data structures.

The numerics library (Clause 26) provides numeric algorithms and complex number components that extend support for numeric processing. The valarray component provides support for n-at-a-time processing, potentially implemented as parallel operations on platforms that support such processing. The random number component provides facilities for generating pseudo-random numbers.
The input/output library (Clause 29) provides the iostream components that are the primary mechanism for C++ program input and output. They can be used with other elements of the library, particularly strings, locales, and iterators.

The regular expressions library (Clause 30) provides regular expression matching and searching.

The atomic operations library (Clause 31) allows more fine-grained concurrent access to shared data than is possible with locks.

The thread support library (Clause 32) provides components to create and manage threads, including mutual exclusion and interthread communication.

16.2 The C standard library

The C++ standard library also makes available the facilities of the C standard library, suitably adjusted to ensure static type safety.

The descriptions of many library functions rely on the C standard library for the semantics of those functions. In some cases, the signatures specified in this document may be different from the signatures in the C standard library, and additional overloads may be declared in this document, but the behavior and the preconditions (including any preconditions implied by the use of an ISO C restrict qualifier) are the same unless otherwise stated.

16.3 Method of description

16.3.1 General

1 Subclause 16.3 describes the conventions used to specify the C++ standard library. 16.3.2 describes the structure of Clause 17 through Clause 32 and Annex D. 16.3.3 describes other editorial conventions.

16.3.2 Structure of each clause

16.3.2.1 Elements

Each library clause contains the following elements, as applicable:

(1.1) — Summary
(1.2) — Requirements
(1.3) — Detailed specifications
(1.4) — References to the C standard library

16.3.2.2 Summary

The Summary provides a synopsis of the category, and introduces the first-level subclauses. Each subclause also provides a summary, listing the headers specified in the subclause and the library entities provided in each header.

16.3.2.3 Requirements

Requirements describe constraints that shall be met by a C++ program that extends the standard library. Such extensions are generally one of the following:

(1.1) — Template arguments
(1.2) — Derived classes

156) To save space, items that do not apply to a Clause are omitted. For example, if a Clause does not specify any requirements, there will be no “Requirements” subclause.
Containers, iterators, and algorithms that meet an interface convention or model a concept

The string and istream components use an explicit representation of operations required of template arguments. They use a class template `char_traits` to define these constraints.

Interface convention requirements are stated as generally as possible. Instead of stating “class X has to define a member function `operator++()`”, the interface requires “for any object x of class X, ++x is defined”. That is, whether the operator is a member is unspecified.

Requirements are stated in terms of well-defined expressions that define valid terms of the types that meet the requirements. For every set of well-defined expression requirements there is either a named concept or a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm (Clause 25) that uses the well-defined expression requirements is described in terms of the valid expressions for its template parameter types.

The library specification uses a typographical convention for naming requirements. Names in italic type that begin with the prefix Cpp17 refer to sets of well-defined expression requirements typically presented in tabular form, possibly with additional prose semantic requirements. For example, Cpp17Destructible (Table 32) is such a named requirement. Names in constant width type refer to library concepts which are presented as a concept definition (Clause 13), possibly with additional prose semantic requirements. For example, destructible (18.4.10) is such a named requirement.

Template argument requirements are sometimes referenced by name. See 16.3.3.3.

In some cases the semantic requirements are presented as C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented.\footnote{Although in some cases the code given is unambiguously the optimum implementation.}

Required operations of any concept defined in this document need not be total functions; that is, some arguments to a required operation may result in the required semantics failing to be met.

\[\text{Example 1: The required} \ < \text{operator of the} \ totally\_ordered \text{concept (18.5.4) does not meet the semantic requirements of that concept when operating on NaNs.} \] \text{— end example}\]

This does not affect whether a type models the concept.

A declaration may explicitly impose requirements through its associated constraints (13.5.3). When the associated constraints refer to a concept (13.7.9), the semantic constraints specified for that concept are additionally imposed on the use of the declaration.

16.3.2.4 Detailed specifications

The detailed specifications each contain the following elements:

1. \(\text{—} \) name and brief description
2. \(\text{—} \) synopsis (class definition or function declaration, as appropriate)
3. \(\text{—} \) restrictions on template arguments, if any
4. \(\text{—} \) description of class invariants
5. \(\text{—} \) description of function semantics

- Descriptions of class member functions follow the order (as appropriate):\footnote{To save space, items that do not apply to a class are omitted. For example, if a class does not specify any comparison operator functions, there will be no “Comparison operator functions” subclause.}
  1. \(\text{—} \) constructor(s) and destructor
  2. \(\text{—} \) copying, moving & assignment functions
  3. \(\text{—} \) comparison operator functions
  4. \(\text{—} \) modifier functions
  5. \(\text{—} \) observer functions
  6. \(\text{—} \) operators and other non-member functions

- Descriptions of function semantics contain the following elements (as appropriate):\footnote{To save space, elements that do not apply to a function are omitted. For example, if a function specifies no preconditions, there will be no Preconditions: element.}

\[\text{§ 16.3.2.4} \quad 480\]
3.1 Constraints: the conditions for the function’s participation in overload resolution (12.4).

[Note 1: Failure to meet such a condition results in the function’s silent non-viability. — end note]

[Example 1: An implementation might express such a condition via a constraint-expression (13.5.3). — end example]

3.2 Mandates: the conditions that, if not met, render the program ill-formed.

[Example 2: An implementation might express such a condition via the constant-expression in a static_assert-declaration (9.1). If the diagnostic is to be emitted only after the function has been selected by overload resolution, an implementation might express such a condition via a constraint-expression (13.5.3) and also define the function as deleted. — end example]

3.3 Preconditions: the conditions that the function assumes to hold whenever it is called; violation of any preconditions results in undefined behavior.

3.4 Effects: the actions performed by the function.

3.5 Synchronization: the synchronization operations (6.9.2) applicable to the function.

3.6 Postconditions: the conditions (sometimes termed observable results) established by the function.

3.7 Returns: a description of the value(s) returned by the function.

3.8 Throws: any exceptions thrown by the function, and the conditions that would cause the exception.

3.9 Complexity: the time and/or space complexity of the function.

3.10 Remarks: additional semantic constraints on the function.

3.11 Error conditions: the error conditions for error codes reported by the function.

Whenever the Effects element specifies that the semantics of some function F are Equivalent to some code sequence, then the various elements are interpreted as follows. If F’s semantics specifies any Constraints or Mandates elements, then those requirements are logically imposed prior to the equivalent-to semantics. Next, the semantics of the code sequence are determined by the Constraints, Mandates, Preconditions, Effects, Synchronization, Postconditions, Returns, Throws, Complexity, Remarks, and Error conditions specified for the function invocations contained in the code sequence. The value returned from F is specified by F’s Returns element, or if F has no Returns element, a non-void return from F is specified by the return statements (8.7.4) in the code sequence. If F’s semantics contains a Throws, Postconditions, or Complexity element, then that supersedes any occurrences of that element in the code sequence.

For non-reserved replacement and handler functions, Clause 17 specifies two behaviors for the functions in question: their required and default behavior. The default behavior describes a function definition provided by the implementation. The required behavior describes the semantics of a function definition provided by either the implementation or a C++ program. Where no distinction is explicitly made in the description, the behavior described is the required behavior.

If the formulation of a complexity requirement calls for a negative number of operations, the actual requirement is zero operations.

Complexity requirements specified in the library clauses are upper bounds, and implementations that provide better complexity guarantees meet the requirements.

Error conditions specify conditions where a function may fail. The conditions are listed, together with a suitable explanation, as the enum class errc constants (19.5).

16.3.2.5 C library [structure.see.also]

16.3.3 Other conventions [conventions]

16.3.3.1 General [conventions.general]

Subclause 16.3.3 describes several editorial conventions used to describe the contents of the C++ standard library. These conventions are for describing implementation-defined types (16.3.3.3), and member functions (16.3.3.4).

1 This simplifies the presentation of complexity requirements in some cases.
16.3.3.2 Exposition-only functions  

Several function templates defined in Clause 17 through Clause 32 and Annex D are only defined for the purpose of exposition. The declaration of such a function is followed by a comment ending in *exposition only*.

The following are defined for exposition only to aid in the specification of the library:

```cpp
template<class T> constexpr decay_t<T> decay-copy(T&& v) 
noexcept(is_nothrow_convertible_v<T, decay_t<T>>) // exposition only
{ return std::forward<T>(v); }

constexpr auto synth-three-way =
[](<class T, class U>(const T& t, const U& u) 
requires requires {
{ t < u } -> boolean-testable;
{ u < t } -> boolean-testable;
}
{ if constexpr (three_way_comparable_with<T, U>) {
  return t <=> u;
} else {
  if (t < u) return weak_ordering::less;
  if (u < t) return weak_ordering::greater;
  return weak_ordering::equivalent;
}
};

template<class T, class U=T>
using synth-three-way-result = decltype(synth-three-way(declval<T&>(), declval<U&>()));
```

16.3.3.3 Type descriptions

16.3.3.3.1 General

The Requirements subclauses may describe names that are used to specify constraints on template arguments. These names are used in library Clauses to describe the types that may be supplied as arguments by a C++ program when instantiating template components from the library.

Certain types defined in Clause 29 are used to describe implementation-defined types. They are based on other types, but with added constraints.

16.3.3.3.2 Exposition-only types

Several types defined in Clause 17 through Clause 32 and Annex D are defined for the purpose of exposition. The declaration of such a type is followed by a comment ending in *exposition only*.

```cpp
namespace std {
  extern "C" using some-handler = int(int, void*, double); // exposition only
}
```

The type placeholder `some-handler` can now be used to specify a function that takes a callback parameter with C language linkage. —end example]

16.3.3.3.3 Enumerated types

Several types defined in Clause 29 are enumerated types. Each enumerated type may be implemented as an enumeration or as a synonym for an enumeration.

The enumerated type `enumerated` can be written:

```cpp
enum enumerated { V0, V1, V2, V3, ... };
```

---

161) Examples from 16.4.4 include: `Cpp17EqualityComparable`, `Cpp17LessThanComparable`, `Cpp17CopyConstructible`. Examples from 23.3 include: `Cpp17InputIterator`, `Cpp17ForwardIterator`.

162) Such as an integer type, with constant integer values (6.8.2).
Here, the names \(C_0, C_1, \ldots\) represent enumerated elements for this particular enumerated type. All such elements have distinct values.

**16.3.3.3.4 Bitmask types**

Several types defined in Clause 17 through Clause 32 and Annex D are bitmask types. Each bitmask type can be implemented as an enumerated type that overloads certain operators, as an integer type, or as a `bitset` (20.9.2).

The bitmask type `bitmask` can be written:

```c
// For exposition only.
// `int_type` is an integral type capable of representing all values of the bitmask type.
enum bitmask : int_type {
  V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, ...
};
```

```c
inline constexpr bitmask C0(V0);
inline constexpr bitmask C1(V1);
inline constexpr bitmask C2(V2);
inline constexpr bitmask C3(V3);
```

```c
constexpr bitmask operator&(bitmask X, bitmask Y) {
  return static_cast<bitmask>(static_cast<int_type>(X) & static_cast<int_type>(Y));
}
```

```c
constexpr bitmask operator|(bitmask X, bitmask Y) {
  return static_cast<bitmask>(static_cast<int_type>(X) | static_cast<int_type>(Y));
}
```

```c
constexpr bitmask operator^(bitmask X, bitmask Y) {
  return static_cast<bitmask>(static_cast<int_type>(X) ^ static_cast<int_type>(Y));
}
```

```c
constexpr bitmask operator~(bitmask X) {
  return static_cast<bitmask>(~static_cast<int_type>(X));
}
```

```c
bitmask& operator&=(bitmask& X, bitmask Y) {
  X = X & Y; return X;
}
```

```c
bitmask& operator|=(bitmask& X, bitmask Y) {
  X = X | Y; return X;
}
```

```c
bitmask& operator^=(bitmask& X, bitmask Y) {
  X = X ^ Y; return X;
}
```

Here, the names \(C_0, C_1, \ldots\) represent bitmask elements for this particular bitmask type. All such elements have distinct, nonzero values such that, for any pair \(C_i\) and \(C_j\) where \(i \neq j\), \(C_i \& C_j\) is nonzero and \(C_i \& C_j\) is zero. Additionally, the value 0 is used to represent an empty bitmask, in which no bitmask elements are set.

The following terms apply to objects and values of bitmask types:

- **(4.1)** To set a value \(Y\) in an object \(X\) is to evaluate the expression \(X \|= Y\).
- **(4.2)** To clear a value \(Y\) in an object \(X\) is to evaluate the expression \(X \&= \sim Y\).
- **(4.3)** The value \(Y\) is set in the object \(X\) if the expression \(X \& Y\) is nonzero.

**16.3.3.5 Character sequences**

**16.3.3.5.1 General**

The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:
A letter is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.

The decimal-point character is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clause 17 through Clause 32 and Annex D by a period, '.', which is also its value in the "C" locale, but may change during program execution by a call to setlocale(int, const char*),163 or by a change to a locale object, as described in 28.3 and Clause 29.

A character sequence is an array object (9.3.4.5) \( A \) that can be declared as \( T A[N] \), where \( T \) is any of the types char, unsigned char, or signed char (6.8.2), optionally qualified by any combination of const or volatile. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value \( S \) that points to its first element.

A null-terminated byte string, or NTBS, is a character sequence whose highest-addressed element with defined content has the value zero (the terminating null character); no other element in the sequence has the value zero.164

The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.

The value of an NTBS is the sequence of values of the elements up to and including the terminating null character.

A static NTBS is an NTBS with static storage duration.165

A null-terminated multibyte string, or NTMBS, is an NTBS that constitutes a sequence of valid multibyte characters, beginning and ending in the initial shift state.166

A static NTMBS is an NTMBS with static storage duration.

A customization point object is a function object (20.14) with a literal class type that interacts with program-defined types while enforcing semantic requirements on that interaction.

The type of a customization point object, ignoring cv-qualifiers, shall model semiregular (18.6).

All instances of a specific customization point object type shall be equal (18.2).

The type \( T \) of a customization point object shall model invocable<const T&, Args...> (18.7.2) when the types in Args... meet the requirements specified in that customization point object’s definition. When the types of Args... do not meet the customization point object’s requirements, \( T \) shall not have a function call operator that participates in overload resolution.

Each customization point object type constrains its return type to model a particular concept.

[Note 1: Many of the customization point objects in the library evaluate function call expressions with an unqualified name which results in a call to a program-defined function found by argument dependent name lookup (6.5.3). To preclude such an expression resulting in a call to unconstrained functions with the same name in namespace std, customization point objects specify that lookup for these expressions is performed in a context that includes deleted overloads matching the signatures of overloads defined in namespace std. When the deleted overloads are viable, program-defined overloads need be more specialized (13.7.7.3) or more constrained (13.5.5) to be used by a customization point object. — end note]

For the sake of exposition, Clause 17 through Clause 32 and Annex D do not describe copy/move constructors, assignment operators, or (non-virtual) destructors with the same apparent semantics as those that can be

---

163) declared in `<locale>` (28.5.1).

164) Many of the objects manipulated by function signatures declared in `<cstring>` (21.5.3) are character sequences or NTBSs. The size of some of these character sequences is limited by a length value, maintained separately from the character sequence.

165) A string-literal, such as "abc", is a static NTBS.

166) An NTBS that contains characters only from the basic execution character set is also an NTMBS. Each multibyte character then consists of a single byte.
generated by default (11.4.5.3, 11.4.6, 11.4.7). It is unspecified whether the implementation provides explicit
definitions for such member function signatures, or for virtual destructors that can be generated by default.

16.3.3.5 Private members

Clause 17 through Clause 32 and Annex D do not specify the representation of classes, and intentionally omit
specification of class members (11.4). An implementation may define static or non-static class members, or
both, as needed to implement the semantics of the member functions specified in Clause 17 through Clause
32 and Annex D.

For the sake of exposition, some subclauses provide representative declarations, and semantic requirements,
for private members of classes that meet the external specifications of the classes. The declarations for such
members are followed by a comment that ends with exposition only, as in:

```cpp
streambuf* sb; // exposition only
```

An implementation may use any technique that provides equivalent observable behavior.

16.4 Library-wide requirements

16.4.1 General

Subclause 16.4 specifies requirements that apply to the entire C++ standard library. Clause 17 through
Clause 32 and Annex D specify the requirements of individual entities within the library.

Requirements specified in terms of interactions between threads do not apply to programs having only a
single thread of execution.

16.4.2 describes the library’s contents and organization, 16.4.3 describes how well-formed C++ programs gain
access to library entities, 16.4.4 describes constraints on types and functions used with the C++ standard
library, 16.4.5 describes constraints on well-formed C++ programs, and 16.4.6 describes constraints on
conforming implementations.

16.4.2 Library contents and organization

16.4.2.1 General

16.4.2.2 describes the entities and macros defined in the C++ standard library. 16.4.2.3 lists the standard library
headers and some constraints on those headers. 16.4.2.4 lists requirements for a freestanding implementation
of the C++ standard library.

16.4.2.2 Library contents

The C++ standard library provides definitions for the entities and macros described in the synopses of the
C++ standard library headers (16.4.2.3), unless otherwise specified.

All library entities except `operator new` and `operator delete` are defined within the namespace `std` or
namespaces nested within namespace `std`. It is unspecified whether names declared in a specific namespace
are declared directly in that namespace or in an inline namespace inside that namespace.

Whenever a name `x` defined in the standard library is mentioned, the name `x` is assumed to be fully qualified as `::std::x`, unless explicitly described otherwise. For example, if the `Effects` element for library function `F` is described as calling library function `G`, the function `::std::G` is meant.

16.4.2.3 Headers

Each element of the C++ standard library is declared or defined (as appropriate) in a `header`.

The C++ standard library provides the C++ library headers, shown in Table 21.

The facilities of the C standard library are provided in the additional headers shown in Table 22.

The headers listed in Table 21, or, for a freestanding implementation, the subset of such headers that are
provided by the implementation, are collectively known as the importable C++ library headers.

[Note 1: Importable C++ library headers can be imported as module units (10.3). — end note]

---

167) The C standard library headers (D.10) also define names within the global namespace, while the C++ headers for C library
facilities (16.4.2.3) can also define names within the global namespace.

168) This gives implementers freedom to use inline namespaces to support multiple configurations of the library.

169) A header is not necessarily a source file, nor are the sequences delimited by `<` and `>` in header names necessarily valid source
file names (15.3).

170) It is intentional that there is no C++ header for any of these C headers: `<stdatomic.h>`, `<stdnoreturn.h>`, `<threads.h>`.
Table 21: C++ library headers  

| `<algorithm>` | `<forward_list>` | `<numbers>` | `<string>` |
| `<any>` | `<fstream>` | `<numeric>` | `<string_view>` |
| `<array>` | `<functional>` | `<optional>` | `<strstream>` |
| `<atomic>` | `<future>` | `<ostream>` | `<syncstream>` |
| `<barrier>` | `<initializer_list>` | `<queue>` | `<system_error>` |
| `<bit>` | `<iosmanip>` | `<random>` | `<thread>` |
| `<bitset>` | `<ios>` | `<ranges>` | `<tuple>` |
| `<chrono>` | `<iosfwd>` | `<ratio>` | `<typeid>` |
| `<codecvt>` | `<iostream>` | `<regex>` | `<typeinfo>` |
| `<compare>` | `<iostreamstream>` | `<scoped_allocator>` | `<type_traits>` |
| `<complex>` | `<iterator>` | `<semaphore>` | `<unordered_map>` |
| `<concepts>` | `<latch>` | `<set>` | `<unordered_set>` |
| `<condition_variable>` | `<limits>` | `<shared_mutex>` | `<utility>` |
| `<coroutine>` | `<list>` | `<source_location>` | `<valarray>` |
| `<deque>` | `<locale>` | `<span>` | `<vector>` |
| `<exception>` | `<longdouble>` | `<sstream>` | `<version>` |
| `<execution>` | `<memory>` | `<string>` | `<wcharchar>` |
| `<filesystem>` | `<mutex>` | `<stop_token>` | `<wctype>` |
| `<format>` | `<new>` | `<streambuf>` | |

Table 22: C++ headers for C library facilities  

| `<cassert>` | `<cfenv>` | `<climits>` | `<csetjmp>` | `<cstddef>` | `<cuchar>` |
| `<cassert>` | `<cfloat>` | `<cinttypes>` | `<cstddef>` | `<cstdlib>` | `<cstring>` |
| `<cctype>` | `<cfenv>` | `<climits>` | `<csetjmp>` | `<cstddef>` | `<ctime>` |
| `<cerrno>` | `<cfloat>` | `<clocale>` | `<csetjmp>` | `<cstring>` | `<cwchar>` |
| `<ctime>` | `<cstring>` | `<ctime>` | `<csetjmp>` | `<cstring>` | `<cwctype>` |

---

**Example 1:**

```cpp
import <vector>;  // imports the <vector> header unit
std::vector<int> vi;  // OK
@end example
```

5 Except as noted in Clause 16 through Clause 32 and Annex D, the contents of each header `cname` is the same as that of the corresponding header `name.h` as specified in the C standard library (Clause 2). In the C++ standard library, however, the declarations (except for names which are defined as macros in C) are within namespace scope (6.4.6) of the namespace `std`. It is unspecified whether these names (including any overloads added in Clause 17 through Clause 32 and Annex D) are first declared within the global namespace scope and are then injected into namespace `std` by explicit using-declarations (9.9).

6 Names which are defined as macros in C shall be defined as macros in the C++ standard library, even if C grants license for implementation as functions.

[Note 2: The names defined as macros in C include the following: assert, offsetof, setjmp, va_arg, va_end, and va_start.  — end note]

7 Names that are defined as functions in C shall be defined as functions in the C++ standard library.\(^{171}\)

8 Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library headers.\(^{172}\)

9 D.10, C standard library headers, describes the effects of using the `name.h` (C header) form in a C++ program.\(^{173}\)

---

\(^{171}\) This disallows the practice, allowed in C, of providing a masking macro in addition to the function prototype. The only way to achieve equivalent inline behavior in C++ is to provide a definition as an extern inline function.

\(^{172}\) In particular, including the standard header `<iso646.h>` has no effect.

\(^{173}\) The `.h` headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace `std`. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.
Annex K of the C standard describes a large number of functions, with associated types and macros, which “promote safer, more secure programming” than many of the traditional C library functions. The names of the functions have a suffix of _s, most of them provide the same service as the C library function with the unsuffixed name, but generally take an additional argument whose value is the size of the result array. If any C++ header is included, it is implementation-defined whether any of these names is declared in the global namespace. (None of them is declared in namespace std.)

Table 23 lists the Annex K names that may be declared in some header. These names are also subject to the restrictions of 16.4.5.3.3.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort_handler_s</td>
<td>mbstowcs_s</td>
<td>strncat_s</td>
<td>vswstringf_s</td>
</tr>
<tr>
<td>asctime_s</td>
<td>memcpy_s</td>
<td>strncpy_s</td>
<td>vwpprintf_s</td>
</tr>
<tr>
<td>bsearch_s</td>
<td>memmove_s</td>
<td>strtok_s</td>
<td>vwscanf_s</td>
</tr>
<tr>
<td>constraint_handler_t</td>
<td>memset_s</td>
<td>swprintf_s</td>
<td>wcrtomb_s</td>
</tr>
<tr>
<td>ctime_s</td>
<td>printf_s</td>
<td>swscanf_s</td>
<td>wcscat_s</td>
</tr>
<tr>
<td>errno_t</td>
<td>qsort_s</td>
<td>tmpfile_s</td>
<td>wcscpy_s</td>
</tr>
<tr>
<td>fopen_s</td>
<td>RSIZE_MAX</td>
<td>tmpnam_s</td>
<td>wcscncat_s</td>
</tr>
<tr>
<td>fprintf_s</td>
<td>size_t</td>
<td>tmpnam_s</td>
<td>wcscncpy_s</td>
</tr>
<tr>
<td>freopen_s</td>
<td>rsize_t</td>
<td>vfprintf_s</td>
<td>wcsmlens_s</td>
</tr>
<tr>
<td>fscanf_s</td>
<td>setconstraint_handler_s</td>
<td>vsfscanf_s</td>
<td>wcstombs_s</td>
</tr>
<tr>
<td>fswprintf_s</td>
<td>snprintf_s</td>
<td>vsfprintf_s</td>
<td>wcstok_s</td>
</tr>
<tr>
<td>fwsscanf_s</td>
<td>snvprintf_s</td>
<td>vsfwscanf_s</td>
<td>wcstombs_s</td>
</tr>
<tr>
<td>getenv_s</td>
<td>sprintf_s</td>
<td>vsprintf_s</td>
<td>wcctomb_s</td>
</tr>
<tr>
<td>gets_s</td>
<td>sscanf_s</td>
<td>vsscanf_s</td>
<td>wmemcpy_s</td>
</tr>
<tr>
<td>gmtime_s</td>
<td>strcat_s</td>
<td>vsscanf_s</td>
<td>wmemmove_s</td>
</tr>
<tr>
<td>ignore_handler_s</td>
<td>strncpy_s</td>
<td>vsnprintf_s</td>
<td>wmemmove_s</td>
</tr>
<tr>
<td>localtime_s</td>
<td>strerrorlen_s</td>
<td>vsnprintf_s</td>
<td>wmemmove_s</td>
</tr>
<tr>
<td>L_tmpnam_s</td>
<td>strerror_s</td>
<td>vsnprintf_s</td>
<td>wmemmove_s</td>
</tr>
<tr>
<td>mbstowcs_s</td>
<td>strlen_s</td>
<td>vsprintf_s</td>
<td>wmemmove_s</td>
</tr>
<tr>
<td>wchar_t_s</td>
<td>vs_Printf_s</td>
<td>wchar_t_s</td>
<td>wmemmove_s</td>
</tr>
</tbody>
</table>

10. Table 23 lists the Annex K names that may be declared in some header. These names are also subject to the restrictions of 16.4.5.3.3.

16.4.2.4 Freestanding implementations

1. Two kinds of implementations are defined: hosted and freestanding (4.1); the kind of the implementation is implementation-defined. For a hosted implementation, this document describes the set of available headers.

2. A freestanding implementation has an implementation-defined set of headers. This set shall include at least the headers shown in Table 24.

3. The supplied version of the header <cstdlib> (17.2.2) shall declare at least the functions abort, atexit, at_quick_exit, exit, and quick_exit (17.5). The supplied version of the header <atomic> (31.2) shall meet the same requirements as for a hosted implementation except that support for always lock-free integral atomic types (31.5) is implementation-defined, and whether or not the type aliases atomic_signed_lock_free and atomic_unsigned_lock_free are defined (31.3) is implementation-defined. The other headers listed in this table shall meet the same requirements as for a hosted implementation.

16.4.3 Using the library

16.4.3.1 Overview

1. Subclause 16.4.3 describes how a C++ program gains access to the facilities of the C++ standard library. 16.4.3.2 describes effects during translation phase 4, while 16.4.3.3 describes effects during phase 8 (5.2).

16.4.3.2 Headers

1. The entities in the C++ standard library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate #include preprocessing directive (15.3) or the appropriate import declaration (10.3).

2. A translation unit may include library headers in any order (5.1). Each may be included more than once, with no effect different from being included exactly once, except that the effect of including either <cassert> (19.3.2) or <assert.h> (D.10) depends each time on the lexically current definition of NDEBUG.174

174 This is the same as the C standard library.
### Table 24: C++ headers for freestanding implementations  
[tab:headers.cpp.fs]

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.2 Types</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>17.3 Implementation</td>
<td><code>&lt;cfloat&gt;</code>, <code>&lt;climits&gt;</code>, <code>&lt;limits&gt;</code>, <code>&lt;version&gt;</code></td>
</tr>
<tr>
<td>17.4 Integer types</td>
<td><code>&lt;cstdint&gt;</code></td>
</tr>
<tr>
<td>17.5 Start and termination</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>17.6 Dynamic memory</td>
<td><code>&lt;new&gt;</code></td>
</tr>
<tr>
<td>17.7 Type identification</td>
<td><code>&lt;typeinfo&gt;</code></td>
</tr>
<tr>
<td>17.8 Source location</td>
<td><code>&lt;source_location&gt;</code></td>
</tr>
<tr>
<td>17.9 Exception handling</td>
<td><code>&lt;exception&gt;</code></td>
</tr>
<tr>
<td>17.10 Initializer lists</td>
<td><code>&lt;initializer_list&gt;</code></td>
</tr>
<tr>
<td>17.11 Comparisons</td>
<td><code>&lt;compare&gt;</code></td>
</tr>
<tr>
<td>17.12 Coroutines support</td>
<td><code>&lt;coroutine&gt;</code></td>
</tr>
<tr>
<td>17.13 Other runtime support</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>Clause 18 Concepts library</td>
<td><code>&lt;concepts&gt;</code></td>
</tr>
<tr>
<td>20.15 Type traits</td>
<td><code>&lt;type_traits&gt;</code></td>
</tr>
<tr>
<td>26.5 Bit manipulation</td>
<td><code>&lt;bit&gt;</code></td>
</tr>
<tr>
<td>Clause 31 Atomics</td>
<td><code>&lt;atomic&gt;</code></td>
</tr>
</tbody>
</table>

3 A translation unit shall include a header only outside of any declaration or definition and, in the case of a module unit, only in its `global-module-fragment`, and shall include the header or import the corresponding header unit lexically before the first reference in that translation unit to any of the entities declared in that header. No diagnostic is required.

### 16.4.3.3 Linkage

1 Entities in the C++ standard library have external linkage (6.6). Unless otherwise specified, objects and functions have the default `extern "C++"` linkage (9.11).

2 Whether a name from the C standard library declared with external linkage has `extern "C"` or `extern "C++"` linkage is implementation-defined. It is recommended that an implementation use `extern "C++"` linkage for this purpose.\(^\text{175}\)

3 Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

4 See also replacement functions (16.4.5.6), runtime changes (16.4.5.7).

### 16.4.4 Requirements on types and expressions

1 **General**

16.4.4.2 describes requirements on types and expressions used to instantiate templates defined in the C++ standard library. 16.4.4.3 describes the requirements on swappable types and swappable expressions. 16.4.4.4 describes the requirements on pointer-like types that support null values. 16.4.4.5 describes the requirements on hash function objects. 16.4.4.6 describes the requirements on storage allocators.

### 16.4.4.2 Template argument requirements

1 The template definitions in the C++ standard library refer to various named requirements whose details are set out in Tables 25–32. In these tables, `{\it T}` is an object or reference type to be supplied by a C++ program instantiating a template; `{\it a}`, `{\it b}`, and `{\it c}` are values of type (possibly `{\it const}`) `{\it T}`; `{\it s}` and `{\it t}` are modifiable lvalues of type `{\it T}`; `{\it u}` denotes an identifier; `{\it rv}` is an rvalue of type `{\it T}`; and `{\it v}` is an lvalue of type (possibly `{\it const}`) `{\it T}` or an rvalue of type `{\it const T}`.

2 In general, a default constructor is not required. Certain container class member function signatures specify `{\it T}()` as a default argument. `{\it T}()` shall be a well-defined expression (9.4) if one of those signatures is called using the default argument (9.3.4.7).

175) The only reliable way to declare an object or function signature from the C standard library is by including the header that declares it, notwithstanding the latitude granted in 7.1.4 of the C Standard.
Table 25: Cpp17EqualityComparable requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
<td>convertible to bool</td>
<td>== is an equivalence relation, that is, it has the following properties:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— For all a, a == a.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— If a == b, then b == a.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— If a == b and b == c, then a == c.</td>
</tr>
</tbody>
</table>

Table 26: Cpp17LessThanComparable requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>convertible to bool</td>
<td>&lt; is a strict weak ordering relation (25.8)</td>
</tr>
</tbody>
</table>

Table 27: Cpp17DefaultConstructible requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T t;</td>
<td>object t is default-initialized</td>
</tr>
<tr>
<td>T u{};</td>
<td>object u is value-initialized or aggregate-initialized</td>
</tr>
<tr>
<td>T()</td>
<td>an object of type T is value-initialized or aggregate-initialized</td>
</tr>
<tr>
<td>T{}</td>
<td></td>
</tr>
</tbody>
</table>

Table 28: Cpp17MoveConstructible requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T u = rv;</td>
<td>u is equivalent to the value of rv before the construction</td>
</tr>
<tr>
<td>T(rv)</td>
<td>T(rv) is equivalent to the value of rv before the construction</td>
</tr>
<tr>
<td></td>
<td>rv’s state is unspecified</td>
</tr>
<tr>
<td></td>
<td>[Note 1: rv must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether rv has been moved from or not. — end note]</td>
</tr>
</tbody>
</table>

Table 29: Cpp17CopyConstructible requirements (in addition to Cpp17MoveConstructible)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T u = v;</td>
<td>the value of v is unchanged and is equivalent to u</td>
</tr>
<tr>
<td>T(v)</td>
<td>the value of v is unchanged and is equivalent to T(v)</td>
</tr>
</tbody>
</table>

16.4.4.3 Swappable requirements

1 This subclause provides definitions for swappable types and expressions. In these definitions, let t denote an expression of type T, and let u denote an expression of type U.

2 An object t is swappable with an object u if and only if:

1. the expressions swap(t, u) and swap(u, t) are valid when evaluated in the context described below, and
Table 30: Cpp17MoveAssignble requirements  

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Return value</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = rv</td>
<td>T&amp;</td>
<td>t</td>
<td>If t and rv do not refer to the same object, t is equivalent to the value of rv before the assignment.</td>
</tr>
</tbody>
</table>

rv’s state is unspecified.  

[Note 2: rv must still meet the requirements of the library component that is using it, whether or not t and rv refer to the same object. The operations listed in those requirements must work as specified whether rv has been moved from or not. — end note]

Table 31: Cpp17CopyAssignble requirements (in addition to Cpp17MoveAssignble)  

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Return value</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = v</td>
<td>T&amp;</td>
<td>t</td>
<td>t is equivalent to v, the value of v is unchanged</td>
</tr>
</tbody>
</table>

Table 32: Cpp17Destructible requirements  

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>u. ~T()</td>
<td>All resources owned by u are reclaimed, no exception is propagated.</td>
</tr>
</tbody>
</table>

[Note 3: Array types and non-object types are not Cpp17Destructible. — end note]

(2.2) — these expressions have the following effects:

(2.2.1) — the object referred to by t has the value originally held by u and

(2.2.2) — the object referred to by u has the value originally held by t.

The context in which swap(t, u) and swap(u, t) are evaluated shall ensure that a binary non-member function named “swap” is selected via overload resolution (12.4) on a candidate set that includes:

(3.1) — the two swap function templates defined in <utility> (20.2.1) and

(3.2) — the lookup set produced by argument-dependent lookup (6.5.3).

[Note 1: If T and U are both fundamental types or arrays of fundamental types and the declarations from the header <utility> are in scope, the overall lookup set described above is equivalent to that of the qualified name lookup applied to the expression std::swap(t, u) or std::swap(u, t) as appropriate. — end note]

[Note 2: It is unspecified whether a library component that has a swappable requirement includes the header <utility> to ensure an appropriate evaluation context. — end note]

An rvalue or lvalue t is swappable if and only if t is swappable with any rvalue or lvalue, respectively, of type T.

A type X meeting any of the iterator requirements (23.3) meets the Cpp17ValueSwappable requirements if, for any dereferenceable object x of type X, *x is swappable.

[Example 1: User code can ensure that the evaluation of swap calls is performed in an appropriate context under the various conditions as follows:]

```cpp
#include <utility>

// Requires: std::forward<T>(t) shall be swappable with std::forward<U>(u).
template<class T, class U>
void value_swap(T&& t, U&& u) {
  using std::swap;
  swap(std::forward<T>(t), std::forward<U>(u)); // OK: uses "swappable with" conditions
  // for rvalues and lvalues
}
```
// Requires: lvalues of T shall be swappable.

```cpp
template<class T>
void lv_swap(T& t1, T& t2) {
    using std::swap;
    swap(t1, t2); // OK: uses swappable conditions for lvalues of type T
}
```

```cpp
namespace N {
    struct A { int m; }
    struct Proxy { A* a; }
    Proxy proxy(A& a) { return Proxy{ &a }; }

    void swap(A& x, Proxy p) {
        std::swap(x.m, p.a->m);
        // OK: uses context equivalent to swappable conditions for fundamental types
    }
    void swap(Proxy p, A& x) { swap(x, p); } // satisfy symmetry constraint
}
```

```cpp
int main() {
    int i = 1, j = 2;
    lv_swap(i, j);
    assert(i == 2 && j == 1);

    N::A a1 = { 5 }, a2 = { -5 };
    value_swap(a1, proxy(a2));
    assert(a1.m == -5 && a2.m == 5);
}
```

16.4.4.4 **Cpp17NullablePointer** requirements

A **Cpp17NullablePointer** type is a pointer-like type that supports null values. A type P meets the **Cpp17NullablePointer** requirements if:

1. P meets the **Cpp17EqualityComparable**, **Cpp17DefaultConstructible**, **Cpp17CopyConstructible**, **Cpp17CopyAssignable**, and **Cpp17Destructible** requirements,
2. lvalues of type P are swappable (16.4.4.3),
3. the expressions shown in Table 33 are valid and have the indicated semantics, and
4. P meets all the other requirements of this subclause.

A value-initialized object of type P produces the null value of the type. The null value shall be equivalent only to itself. A default-initialized object of type P may have an indeterminate value.

[Note 1: Operations involving indeterminate values might cause undefined behavior. — end note]

An object p of type P can be contextually converted to `bool` (7.3). The effect shall be as if p != nullptr had been evaluated in place of p.

No operation which is part of the **Cpp17NullablePointer** requirements shall exit via an exception.

In Table 33, u denotes an identifier, t denotes a non-const lvalue of type P, a and b denote values of type (possibly const) P, and np denotes a value of type (possibly const) std::nullptr_t.

16.4.4.5 **Cpp17Hash** requirements

A type H meets the **Cpp17Hash** requirements if:

1. it is a function object type (20.14),
2. it meets the **Cpp17CopyConstructible** (Table 29) and **Cpp17Destructible** (Table 32) requirements, and
3. the expressions shown in Table 34 are valid and have the indicated semantics.

Given Key is an argument type for function objects of type H, in Table 34 h is a value of type (possibly const) H, u is an lvalue of type Key, and k is a value of a type convertible to (possibly const) Key.
Table 33: Cpp17NullablePointer requirements  [tab:cpp17.nullablepointer]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>P u(np);</td>
<td></td>
<td>Postconditions: u == nullptr</td>
</tr>
<tr>
<td>P u = np;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P(np)</td>
<td></td>
<td>Postconditions: P(np) == nullptr</td>
</tr>
<tr>
<td>t = np</td>
<td>P&amp;</td>
<td>Postconditions: t == nullptr</td>
</tr>
<tr>
<td>a != b</td>
<td>contextually convertible to bool</td>
<td>!(a == b)</td>
</tr>
<tr>
<td>a == np</td>
<td>contextually convertible to bool</td>
<td>a == P()</td>
</tr>
<tr>
<td>np == a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a != np</td>
<td>contextually convertible to bool</td>
<td>!(a == np)</td>
</tr>
<tr>
<td>np != a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 34: Cpp17Hash requirements  [tab:cpp17.hash]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
</table>
| h(k)       | size_t      | The value returned shall depend only on the argument k for the duration of the program.  
|            |             | [Note 1: Thus all evaluations of the expression h(k) with the same value for k yield the same result for a given execution of the program. —end note] |
|            |             | For two different values t1 and t2, the probability that h(t1) and h(t2) compare equal should be very small, approaching 1.0 / numeric_limits<size_t>::max(). |
| h(u)       | size_t      | Shall not modify u. |

16.4.4.6 Cpp17Allocator requirements  [allocator.requirements]

16.4.4.6.1 General  [allocator.requirements.general]

1 The library describes a standard set of requirements for allocators, which are class-type objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the string types (Clause 21), containers (Clause 22) (except array), string buffers and string streams (Clause 29), and match_results (Clause 30) are parameterized in terms of allocators.

2 The class template allocator_traits (20.10.9) supplies a uniform interface to all allocator types. Table 35 describes the types manipulated through allocators. Table 36 describes the requirements on allocator types and thus on types used to instantiate allocator_traits. A requirement is optional if the last column of Table 36 specifies a default for a given expression. Within the standard library allocator_traits template, an optional requirement that is not supplied by an allocator is replaced by the specified default expression. A user specialization of allocator_traits may provide different defaults and may provide defaults for different requirements than the primary template. Within Tables 35 and 36, the use of move and forward always refers to std::move and std::forward, respectively.

Table 36: Cpp17Allocator requirements  [tab:cpp17.allocator]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::pointer</td>
<td>T*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| X::const_pointer |             | X::pointer is convertible to pointer_-
|                |             | X::const_pointer                |         |
|              |             | pointer_traits<X::const_pointer> |         |
|              |             | rebind<const T>                  |         |
Table 36: `Cpp17Allocator` requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::void_pointer</code></td>
<td><code>X::pointer</code> is convertible to <code>X::void_pointer</code></td>
<td></td>
<td>pointer_-</td>
</tr>
<tr>
<td><code>Y::void_pointer</code></td>
<td><code>X::void_pointer</code>, <code>Y::void_pointer</code></td>
<td></td>
<td>traits&lt;X::&gt;</td>
</tr>
<tr>
<td></td>
<td><code>X::void_pointer and Y::void_pointer</code> are the same type.</td>
<td></td>
<td>pointer&gt;::</td>
</tr>
<tr>
<td><code>X::const_void_pointer</code></td>
<td><code>X::pointer</code>, <code>X::const_pointer</code>, <code>X::void_pointer</code></td>
<td></td>
<td>pointer_-</td>
</tr>
<tr>
<td><code>Y::const_void_pointer</code></td>
<td><code>X::void_pointer are convertible to</code></td>
<td></td>
<td>traits&lt;X::&gt;</td>
</tr>
<tr>
<td></td>
<td><code>X::const_void_pointer</code>, <code>X::const_void_pointer and Y::const_void_pointer</code></td>
<td></td>
<td>rebind&lt;void&gt;</td>
</tr>
<tr>
<td><code>X::value_type</code></td>
<td>Identical to <code>T</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>X::size_type</code></td>
<td>unsigned integer type</td>
<td>a type that can represent the size of the largest object in the allocation model</td>
<td>make_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t&lt;X::&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>difference_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>type&gt;</td>
</tr>
<tr>
<td><code>X::difference_type</code></td>
<td>signed integer type</td>
<td>a type that can represent the difference between any two pointers in the allocation model</td>
<td>pointer_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>traits&lt;X::&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>rebind&lt;void&gt;</td>
</tr>
<tr>
<td><code>typename X::template rebind&lt;U&gt;::other</code></td>
<td>Y</td>
<td>For all <code>U</code> (including <code>T</code>), <code>Y::template rebind&lt;T&gt;::other is X.</code></td>
<td>See Note A, below.</td>
</tr>
<tr>
<td><code>*p</code> <code>T&amp;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>*q</code> <code>const T&amp;</code></td>
<td>*q refers to the same object as *p.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>p-&gt;m</code> <code>type of T::m</code></td>
<td>Preconditions: <code>(*p).m</code> is well-defined.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>equivalent to <code>(*p).m</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>q-&gt;m</code> <code>type of T::m</code></td>
<td>Preconditions: <code>(*q).m</code> is well-defined.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>equivalent to <code>(*q).m</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>static_cast&lt;X::pointer&gt;(w)</code></td>
<td><code>X::pointer</code></td>
<td><code>static_cast&lt;X::pointer&gt;(w)</code> == <code>p</code></td>
<td></td>
</tr>
<tr>
<td><code>X::pointer</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>static_cast&lt;X::const_pointer&gt;</code></td>
<td><code>X::const_pointer</code></td>
<td><code>static_cast&lt;X::const_pointer&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>X::pointer</code></td>
<td></td>
<td><code>X::const_pointer(x) == q</code></td>
<td></td>
</tr>
<tr>
<td><code>pointer_traits&lt;X::pointer&gt;</code></td>
<td><code>X::pointer</code></td>
<td>same as <code>p</code></td>
<td></td>
</tr>
<tr>
<td><code>X::pointer</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>::pointer_to(r)</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

§ 16.4.4.6.1
Table 36: *Cpp17Allocator* requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Default</th>
</tr>
</thead>
</table>
| a.allocate(n) | X::pointer | Memory is allocated for an array of n T and such an object is created but array elements are not constructed. [Example 1: When reusing storage denoted by some pointer value p, 
\[\text{launer} \text{(reinterpret}_{-}\text{cast}\text{(<T*>(new (p) byte[n * sizeof(T)])))} \text{can be used to implicitly create a suitable array object and obtain a pointer to it.}
\]-end example] allocate may throw an appropriate exception.176 |

| | | | |
| a.allocate(n, y) | X::pointer | Same as a.allocate(n). The use of y is unspecified, but it is intended as an aid to locality. a.allocate(n) |
| a.deallocate(p,n) | (not used) | Preconditions: p is a value returned by an earlier call to allocate that has not been invalidated by an intervening call to deallocate. n matches the value passed to allocate to obtain this memory. Throws: Nothing. |
| a.max_size() | X::size_type | the largest value that can meaningfully be passed to X::allocate() numeric_limits<size_type>::max() / sizeof(value_type) |
| ai == a2 | bool | Returns true only if storage allocated from each can be deallocated via the other. operator== shall be reflexive, symmetric, and transitive, and shall not exit via an exception. |
| ai != a2 | bool | same as !(ai == a2) |
| a == b | bool | same as a == Y::rebind<T>::other(b) |
| a != b | bool | same as !(a == b) |
| X u(a); X u = a; | | Shall not exit via an exception. Postconditions: u == a |
| X u(b); | | Shall not exit via an exception. Postconditions: Y(u) == b, u == X(b) |
| X u(std::move(a)); X u = std::move(a); | | Shall not exit via an exception. Postconditions: The value of a is unchanged and is equal to u. |
| X u(std::move(b)); | | Shall not exit via an exception. Postconditions: u is equal to the prior value of X(b). |

176 It is intended that a.allocate be an efficient means of allocating a single object of type T, even when sizeof(T) is small. That is, there is no need for a container to maintain its own free list.
### Table 36: Cpp17Allocator requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.construct(c, args)</td>
<td>(not used)</td>
<td><strong>Effects</strong>: Constructs an object of type C at c.</td>
<td>construct_at(c, std::forward&lt;Args&gt;(args)...).</td>
</tr>
<tr>
<td>a.destroy(c)</td>
<td>(not used)</td>
<td><strong>Effects</strong>: Destroys the object at c</td>
<td>destroy_at(c).</td>
</tr>
<tr>
<td>a.select_on_container_copy_construction()</td>
<td>c</td>
<td>Typically returns either a or X().</td>
<td>return a;</td>
</tr>
<tr>
<td>X::propagate_on_container_copy_assignment</td>
<td>Identical to or derived from true_type or false_type</td>
<td>true_type only if an allocator of type X should be copied when the client container is copy-assigned. See Note B, below.</td>
<td></td>
</tr>
<tr>
<td>X::propagate_on_container_move_assignment</td>
<td>Identical to or derived from true_type or false_type</td>
<td>true_type only if an allocator of type X should be moved when the client container is move-assigned. See Note B, below.</td>
<td></td>
</tr>
<tr>
<td>X::propagate_on_container_swap</td>
<td>Identical to or derived from true_type or false_type</td>
<td>true_type only if an allocator of type X should be swapped when the client container is swapped. See Note B, below.</td>
<td></td>
</tr>
<tr>
<td>X::is_always_equal</td>
<td>Identical to or derived from true_type or false_type</td>
<td>true_type only if the expression a1 == a2 is guaranteed to be true for any two (possibly const) values a1, a2 of type X.</td>
<td></td>
</tr>
</tbody>
</table>

3 Note A: The member class template rebind in the table above is effectively a typedef template.

[Note 2: In general, if the name Allocator is bound to SomeAllocator<T>, then Allocator::rebind<U>::other is the same type as SomeAllocator<U>, where SomeAllocator<T>::value_type is T and SomeAllocator<U>::value_type is U. — end note]

If Allocator is a class template instantiation of the form SomeAllocator<T, Args>, where Args is the zero or more type arguments, and Allocator does not supply a rebind member template, the standard allocator_traits template uses SomeAllocator<T, Args> in place of Allocator::rebind<U>::other by default. For allocator types that are not template instantiations of the above form, no default is provided.

4 Note B: If X::propagate_on_container_copy_assignment::value is true, X shall meet the Cpp17CopyAssignable requirements (Table 31) and the copy operation shall not throw exceptions. If X::propagate_on_container_move_assignment::value is true, X shall meet the Cpp17MoveAssignable requirements (Table 30) and the move operation shall not throw exceptions. If X::propagate_on_container_swap::value is true, lvalues of type X shall be swappable (16.4.4.3) and the swap operation shall not throw exceptions.

An allocator type X shall meet the Cpp17CopyConstructible requirements (Table 29). The X::pointer, X::const_pointer, X::void_pointer, and X::const_void_pointer types shall meet the Cpp17NullablePointer requirements (Table 33). No constructor, comparison operator function, copy operation, move operation, or swap operation on these pointer types shall exit via an exception. X::pointer and X::const_pointer shall also meet the requirements for a Cpp17RandomAccessIterator (23.3.5.7) and the additional requirement that, when a and (a + n) are dereferenceable (16.4.4.3) and the swap operation shall not throw exceptions.

Let x1 and x2 denote objects of (possibly different) types X::void_pointer, X::const_void_pointer, X::pointer, or X::const_pointer. Then, x1 and x2 are equivalently-valued pointer values, if and only if both x1 and x2 can be explicitly converted to the two corresponding objects px1 and px2 of type X::const_void_pointer.

\[
\text{addressof}(\ast (a + n)) = \text{addressof}(\ast a) + n
\]

is true.
Table 35: Descriptive variable definitions  

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T, U, C</td>
<td>any cv-unqualified object type (6.8)</td>
</tr>
<tr>
<td>X</td>
<td>an allocator class for type T</td>
</tr>
<tr>
<td>Y</td>
<td>the corresponding allocator class for type U</td>
</tr>
<tr>
<td>XX</td>
<td>the type allocator_traits&lt;X&gt;</td>
</tr>
<tr>
<td>YY</td>
<td>the type allocator_traits&lt;Y&gt;</td>
</tr>
<tr>
<td>a, a1, a2</td>
<td>lvalues of type X</td>
</tr>
<tr>
<td>u</td>
<td>the name of a variable being declared</td>
</tr>
<tr>
<td>b</td>
<td>a value of type Y</td>
</tr>
<tr>
<td>c</td>
<td>a pointer of type C through which indirection is valid</td>
</tr>
<tr>
<td>p</td>
<td>a value of type XX::pointer, obtained by calling a1.allocate, where a1 == a</td>
</tr>
<tr>
<td>q</td>
<td>a value of type XX::const_pointer obtained by conversion from a value p</td>
</tr>
<tr>
<td>r</td>
<td>a value of type T&amp; obtained by the expression *p</td>
</tr>
<tr>
<td>w</td>
<td>a value of type XX::void_pointer obtained by conversion from a value p</td>
</tr>
<tr>
<td>x</td>
<td>a value of type XX::const_void_pointer obtained by conversion from a value q or a value w</td>
</tr>
<tr>
<td>y</td>
<td>a value of type XX::const_void_pointer obtained by conversion from a result value of YY::allocate, or else a value of type (possibly const) std::nullptr_t</td>
</tr>
<tr>
<td>n</td>
<td>a value of type XX::size_type</td>
</tr>
<tr>
<td>Args</td>
<td>a template parameter pack</td>
</tr>
<tr>
<td>args</td>
<td>a function parameter pack with the pattern Args&lt;kk&gt;</td>
</tr>
</tbody>
</table>

pointer, using a sequence of static_casts using only these four types, and the expression px1 == px2 evaluates to true.

7 Let w1 and w2 denote objects of type X::void_pointer. Then for the expressions

- w1 == w2
- w1 != w2

either or both objects may be replaced by an equivalently-valued object of type X::const_void_pointer with no change in semantics.

8 Let p1 and p2 denote objects of type X::pointer. Then for the expressions

- p1 == p2
- p1 != p2
- p1 < p2
- p1 <= p2
- p1 > p2
- p1 >= p2
- p1 - p2

either or both objects may be replaced by an equivalently-valued object of type X::const_pointer with no change in semantics.

9 An allocator may constrain the types on which it can be instantiated and the arguments for which its construct or destroy members may be called. If a type cannot be used with a particular allocator, the allocator class or the call to construct or destroy may fail to instantiate.

10 If the alignment associated with a specific over-aligned type is not supported by an allocator, instantiation of the allocator for that type may fail. The allocator also may silently ignore the requested alignment.

[Note 3: Additionally, the member function allocate for that type can fail by throwing an object of type bad_alloc. — end note]

11 [Example 2: The following is an allocator class template supporting the minimal interface that meets the requirements of Table 36:]

§ 16.4.4.6.1 496
template<class Tp>
struct SimpleAllocator {
  typedef Tp value_type;
  SimpleAllocator(ctor args);

  template<class T> SimpleAllocator(const SimpleAllocator<T>& other);

  [[nodiscard]] Tp* allocate(std::size_t n);
  void deallocate(Tp* p, std::size_t n);
};

template<class T, class U>
bool operator==(const SimpleAllocator<T>&, const SimpleAllocator<U>&);

16.4.4.6.2 Allocator completeness requirements

If X is an allocator class for type T, X additionally meets the allocator completeness requirements if, whether or not T is a complete type:

1. X is a complete type, and
2. all the member types of allocator_traits<X> (20.10.9) other than value_type are complete types.

16.4.5 Constraints on programs

16.4.5.1 Overview

Subclause 16.4.5 describes restrictions on C++ programs that use the facilities of the C++ standard library. The following subclauses specify constraints on the program’s use of namespaces (16.4.5.2.1), its use of various reserved names (16.4.5.3), its use of headers (16.4.5.4), its use of standard library classes as base classes (16.4.5.5), its definitions of replacement functions (16.4.5.6), and its installation of handler functions during execution (16.4.5.7).

16.4.5.2 Namespace use

16.4.5.2.1 Namespace std

Unless otherwise specified, the behavior of a C++ program is undefined if it adds declarations or definitions to namespace std or to a namespace within namespace std.

Unless explicitly prohibited, a program may add a template specialization for any standard library class template to namespace std provided that (a) the added declaration depends on at least one program-defined type and (b) the specialization meets the standard library requirements for the original template. The behavior of a C++ program is undefined if it declares an explicit or partial specialization of any standard library variable template, except where explicitly permitted by the specification of that variable template.

The behavior of a C++ program is undefined if it declares

1. an explicit specialization of any member function of a standard library class template, or
2. an explicit specialization of any member function template of a standard library class or class template, or
3. an explicit or partial specialization of any member class template of a standard library class or class template, or
4. a deduction guide for any standard library class template.

A program may explicitly instantiate a class template defined in the standard library only if the declaration (a) depends on the name of at least one program-defined type and (b) the instantiation meets the standard library requirements for the original template.

Let F denote a standard library function (16.4.6.4), a standard library static member function, or an instantiation of a standard library function template. Unless F is designated an addressable function, the

---

177) Any library code that instantiates other library templates must be prepared to work adequately with any user-supplied specialization that meets the minimum requirements of this document.
behavior of a C++ program is unspecified (possibly ill-formed) if it explicitly or implicitly attempts to form a pointer to $F$.

[Note 1: Possible means of forming such pointers include application of the unary & operator (7.6.2.2), addressof (20.10.11), or a function-to-pointer standard conversion (7.3.4). — end note]

Moreover, the behavior of a C++ program is unspecified (possibly ill-formed) if it attempts to form a reference to $F$ or if it attempts to form a pointer-to-member designating either a standard library non-static member function (16.4.6.5) or an instantiation of a standard library member function template.

7 Other than in namespace std or in a namespace within namespace std, a program may provide an overload for any library function template designated as a customization point, provided that (a) the overload’s declaration depends on at least one user-defined type and (b) the overload meets the standard library requirements for the customization point.\(^\text{[178]}\)

[Note 2: This permits a (qualified or unqualified) call to the customization point to invoke the most appropriate overload for the given arguments. — end note]

8 A translation unit shall not declare namespace std to be an inline namespace (9.8.2).

16.4.5.2.2 Namespace posix

[namespace.posix]

The behavior of a C++ program is undefined if it adds declarations or definitions to namespace posix or to a namespace within namespace posix unless otherwise specified. The namespace posix is reserved for use by ISO/IEC 9945 and other POSIX standards.

16.4.5.2.3 Namespaces for future standardization

[namespace.future]

Top-level namespaces whose namespace-name consists of std followed by one or more digits (5.10) are reserved for future standardization. The behavior of a C++ program is undefined if it adds declarations or definitions to such a namespace.

[Example 1: The top-level namespace std2 is reserved for use by future revisions of this International Standard. — end example]

16.4.5.3 Reserved names

[reserved.names]

16.4.5.3.1 General

[reserved.names.general]

The C++ standard library reserves the following kinds of names:

(1.1) — macros
(1.2) — global names
(1.3) — names with external linkage

2 If a program declares or defines a name in a context where it is reserved, other than as explicitly allowed by Clause 16, its behavior is undefined.

16.4.5.3.2 Zombie names

[zombie.names]

In namespace std, the following names are reserved for previous standardization:

(1.1) — auto_ptr,
(1.2) — auto_ptr_ref,
(1.3) — binary_function,
(1.4) — binary_negate,
(1.5) — bind1st,
(1.6) — bind2nd,
(1.7) — binder1st,
(1.8) — binder2nd,
(1.9) — const_mem_fun1_ref_t,

\(^{[178]}\) Any library customization point must be prepared to work adequately with any user-defined overload that meets the minimum requirements of this document. Therefore an implementation can elect, under the as-if rule (6.9.1), to provide any customization point in the form of an instantiated function object (20.14) even though the customization point’s specification is in the form of a function template. The template parameters of each such function object and the function parameters and return type of the object’s operator() must match those of the corresponding customization point’s specification.
The following names are reserved as member types for previous standardization, and may not be used as a name for object-like macros in portable code:

(2.1) — argument_type,
(2.2) — first_argument_type,
(2.3) — io_state,
(2.4) — open_mode,
(2.5) — second_argument_type, and
(2.6) — seek_dir.

The name stossc is reserved as a member function for previous standardization, and may not be used as a name for function-like macros in portable code.

The header names <ccomplex>, <ciso646>, <cstdalign>, <cstdbool>, and <ctgmath> are reserved for previous standardization.
16.4.5.3.3 Macro names

1 A translation unit that includes a standard library header shall not \#define or \#undef names declared in any standard library header.

2 A translation unit shall not \#define or \#undef names lexically identical to keywords, to the identifiers listed in Table 4, or to the attribute-tokens described in 9.12, except that the names likely and unlikely may be defined as function-like macros (15.6).

16.4.5.3.4 External linkage

1 Each name declared as an object with external linkage in a header is reserved to the implementation to designate that library object with external linkage, \(^{179}\) both in namespace std and in the global namespace.

2 Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage. \(^{180}\)

3 Each name from the C standard library declared with external linkage is reserved to the implementation for use as a name with extern "C" linkage, both in namespace std and in the global namespace.

4 Each function signature from the C standard library declared with external linkage is reserved to the implementation for use as a function signature with both extern "C" and extern "C++" linkage, \(^{181}\) or as a name of namespace scope in the global namespace.

16.4.5.3.5 Types

1 For each type \(T\) from the C standard library, the types \(::T\) and std::\(T\) are reserved to the implementation and, when defined, \(::T\) shall be identical to std::\(T\).

16.4.5.3.6 User-defined literal suffixes

1 Literal suffix identifiers (12.8) that do not start with an underscore are reserved for future standardization.

16.4.5.4 Headers

1 If a file with a name equivalent to the derived file name for one of the C++ standard library headers is not provided as part of the implementation, and a file with that name is placed in any of the standard places for a source file to be included (15.3), the behavior is undefined.

16.4.5.5 Derived classes

1 Virtual member function signatures defined for a base class in the C++ standard library may be overridden in a derived class defined in the program (11.7.3).

16.4.5.6 Replacement functions

1 Clause 17 through Clause 32 and Annex D describe the behavior of numerous functions defined by the C++ standard library. Under some circumstances, however, certain of these function descriptions also apply to replacement functions defined in the program.

2 A C++ program may provide the definition for any of the following dynamic memory allocation function signatures declared in header <new> (6.7.5.5, 17.6.2):

```cpp
operator new(std::size_t)
operator new(std::size_t, std::align_val_t)
operator new(std::size_t, const std::nothrow_t&)
operator new(std::size_t, std::align_val_t, const std::nothrow_t&)
operator delete(void*)
operator delete(void*, std::size_t)
operator delete(void*, std::align_val_t)
operator delete(void*, std::size_t, std::align_val_t)
operator delete(void*, const std::nothrow_t&)
operator delete(void*, std::align_val_t, const std::nothrow_t&)
```

\(^{179}\) The list of such reserved names includes errno, declared or defined in <cerrno> (19.4.2).

\(^{180}\) The list of such reserved function signatures with external linkage includes setjmp(jmp_buf), declared or defined in <csetjmp> (17.13.3), and va_end(va_list), declared or defined in <cassert> (17.13.2).

\(^{181}\) The function signatures declared in <cuchar> (21.5.5), <cwchar> (21.5.4), and <cwctype> (21.5.2) are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.
The program’s definitions are used instead of the default versions supplied by the implementation (17.6.3). Such replacement occurs prior to program startup (6.3, 6.9.3). The program’s declarations shall not be specified as inline. No diagnostic is required.

16.4.5.7 Handler functions

The C++ standard library provides a default version of the following handler function (Clause 17):

(1.1) — `terminate_handler`

A C++ program may install different handler functions during execution, by supplying a pointer to a function defined in the program or the library as an argument to (respectively):

(2.1) — `set_new_handler`

(2.2) — `set_terminate`

See also subclauses 17.6.4, Storage allocation errors, and 17.9, Exception handling.

A C++ program can get a pointer to the current handler function by calling the following functions:

(3.1) — `get_new_handler`

(3.2) — `get_terminate`

Calling the `set_*` and `get_*` functions shall not incur a data race. A call to any of the `set_*` functions shall synchronize with subsequent calls to the same `set_*` function and to the corresponding `get_*` function.

16.4.5.8 Other functions

In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ standard library depends on components supplied by a C++ program. If these components do not meet their requirements, this document places no requirements on the implementation.

In particular, the effects are undefined in the following cases:

(2.1) — For replacement functions (17.6.3), if the installed replacement function does not implement the semantics of the applicable `Required behavior`: paragraph.

(2.2) — For handler functions (17.6.4.3, 17.9.5.1), if the installed handler function does not implement the semantics of the applicable `Required behavior`: paragraph.

(2.3) — For types used as template arguments when instantiating a template component, if the operations on the type do not implement the semantics of the applicable `Requirements` subclause (16.4.4.6, 22.2, 23.3, 25.2, 26.2). Operations on such types can report a failure by throwing an exception unless otherwise specified.

(2.4) — If any replacement function or handler function or destructor operation exits via an exception, unless specifically allowed in the applicable `Required behavior`: paragraph.

(2.5) — If an incomplete type (6.8) is used as a template argument when instantiating a template component or evaluating a concept, unless specifically allowed for that component.

16.4.5.9 Function arguments

Each of the following applies to all arguments to functions defined in the C++ standard library, unless explicitly stated otherwise.

(1.1) — If an argument to a function has an invalid value (such as a value outside the domain of the function or a pointer invalid for its intended use), the behavior is undefined.
If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

If a function argument binds to an rvalue reference parameter, the implementation may assume that this parameter is a unique reference to this argument.

[Note 1: If the parameter is a generic parameter of the form \(T&&\) and an lvalue of type \(A\) is bound, the argument binds to an lvalue reference (13.10.3.2) and thus is not covered by the previous sentence. — end note]

[Note 2: If a program casts an lvalue to an xvalue while passing that lvalue to a library function (e.g., by calling the function with the argument \(\text{std::move}(x)\)), the program is effectively asking that function to treat that lvalue as a temporary object. The implementation is free to optimize away aliasing checks which might be needed if the argument was an lvalue. — end note]

16.4.5.10 Library object access

The behavior of a program is undefined if calls to standard library functions from different threads may introduce a data race. The conditions under which this may occur are specified in 16.4.6.10.

[Note 1: Modifying an object of a standard library type that is shared between threads risks undefined behavior unless objects of that type are explicitly specified as being shareable without data races or the user supplies a locking mechanism. — end note]

If an object of a standard library type is accessed, and the beginning of the object’s lifetime (6.7.3) does not happen before the access, or the access does not happen before the end of the object’s lifetime, the behavior is undefined unless otherwise specified.

[Note 2: This applies even to objects such as mutexes intended for thread synchronization. — end note]

16.4.5.11 Semantic requirements

A sequence \(\text{Args}\) of template arguments is said to model a concept \(C\) if \(\text{Args}\) satisfies \(C\) (13.5.3) and meets all semantic requirements (if any) given in the specification of \(C\).

If the validity or meaning of a program depends on whether a sequence of template arguments models a concept, and the concept is satisfied but not modeled, the program is ill-formed, no diagnostic required.

If the semantic requirements of a declaration’s constraints (16.3.2.3) are not modeled at the point of use, the program is ill-formed, no diagnostic required.

16.4.6 Conforming implementations

16.4.6.1 Overview

Subclause 16.4.6 describes the constraints upon, and latitude of, implementations of the C++ standard library.

An implementation’s use of headers is discussed in 16.4.6.2, its use of macros in 16.4.6.3, non-member functions in 16.4.6.4, member functions in 16.4.6.5, data race avoidance in 16.4.6.10, access specifiers in 16.4.6.11, class derivation in 16.4.6.12, and exceptions in 16.4.6.13.

16.4.6.2 Headers

A C++ header may include other C++ headers. A C++ header shall provide the declarations and definitions that appear in its synopsis. A C++ header shown in its synopsis as including other C++ headers shall provide the declarations and definitions that appear in the synopses of those other headers.

Certain types and macros are defined in more than one header. Every such entity shall be defined such that any header that defines it may be included after any other header that also defines it (6.3).

The C standard library headers (D.10) shall include only their corresponding C++ standard library header, as described in 16.4.2.3.

16.4.6.3 Restrictions on macro definitions

The names and global function signatures described in 16.4.2.2 are reserved to the implementation.

All object-like macros defined by the C standard library and described in this Clause as expanding to integral constant expressions are also suitable for use in \#if preprocessing directives, unless explicitly stated otherwise.
16.4.6.4 Non-member functions

1 It is unspecified whether any non-member functions in the C++ standard library are defined as inline (9.2.8).
2 A call to a non-member function signature described in Clause 17 through Clause 32 and Annex D shall behave as if the implementation declared no additional non-member function signatures.\footnote{A valid C++ program always calls the expected library non-member function. An implementation can also define additional non-member functions that would otherwise not be called by a valid C++ program.}
3 An implementation shall not declare a non-member function signature with additional default arguments.
4 Unless otherwise specified, calls made by functions in the standard library to non-operator, non-member functions do not use functions from another namespace which are found through argument-dependent name lookup (6.5.3).

[Note 1: The phrase “unless otherwise specified” applies to cases such as the swappable with requirements (16.4.4.3). The exception for overloaded operators allows argument-dependent lookup in cases like that of `ostream_iterator::operator=` (23.6.3.3):
Effects:

```cpp
*out_stream << value;
if (delim != 0)
  *out_stream << delim;
return *this;
```
—end note]

16.4.6.5 Member functions

1 It is unspecified whether any member functions in the C++ standard library are defined as inline (9.2.8).
2 For a non-virtual member function described in the C++ standard library, an implementation may declare a different set of member function signatures, provided that any call to the member function that would select an overload from the set of declarations described in this document behaves as if that overload were selected.

[Note 1: For instance, an implementation can add parameters with default values, or replace a member function with default arguments with two or more member functions with equivalent behavior, or add additional signatures for a member function name. — end note]

16.4.6.6 Friend functions

1 Whenever this document specifies a friend declaration of a function or function template within a class or class template definition, that declaration shall be the only declaration of that function or function template provided by an implementation.

[Note 1: In particular, an implementation is not allowed to provide an additional declaration of that function or function template at namespace scope. — end note]

[Note 2: Such a friend function or function template declaration is known as a hidden friend, as it is visible neither to ordinary unqualified lookup (6.5.2) nor to qualified lookup (6.5.4). — end note]

16.4.6.7 Constexpr functions and constructors

1 This document explicitly requires that certain standard library functions are `constexpr` (9.2.6). An implementation shall not declare any standard library function signature as `constexpr` except for those where it is explicitly required. Within any header that provides any non-defining declarations of constexpr functions or constructors an implementation shall provide corresponding definitions.

16.4.6.8 Requirements for stable algorithms

1 When the requirements for an algorithm state that it is “stable” without further elaboration, it means:

1.1 For the sort algorithms the relative order of equivalent elements is preserved.
1.2 For the remove and copy algorithms the relative order of the elements that are not removed is preserved.
1.3 For the merge algorithms, for equivalent elements in the original two ranges, the elements from the first range (preserving their original order) precede the elements from the second range (preserving their original order).

16.4.6.9 Reentrancy

1 Except where explicitly specified in this document, it is implementation-defined which functions in the C++ standard library may be recursively reentered.
16.4.6.10 Data race avoidance

This subclause specifies requirements that implementations shall meet to prevent data races (6.9.2). Every standard library function shall meet each requirement unless otherwise specified. Implementations may prevent data races in cases other than those specified below.

A C++ standard library function shall not directly or indirectly access objects (6.9.2) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function’s arguments, including this.

A C++ standard library function shall not directly or indirectly modify objects (6.9.2) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function’s non-const arguments, including this.

[Note 1: This means, for example, that implementations can’t use an object with static storage duration for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. — end note]

A C++ standard library function shall not access objects indirectly accessible via its arguments or via elements of its container arguments except by invoking functions required by its specification on those container elements.

Operations on iterators obtained by calling a standard library container or string member function may access the underlying container, but shall not modify it.

[Note 2: In particular, container operations that invalidate iterators conflict with operations on iterators associated with that container. — end note]

Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

Unless otherwise specified, C++ standard library functions shall perform all operations solely within the current thread if those operations have effects that are visible (6.9.2) to users.

[Note 3: This allows implementations to parallelize operations if there are no visible side effects. — end note]

16.4.6.11 Protection within classes

It is unspecified whether any function signature or class described in Clause 17 through Clause 32 and Annex D is a friend of another class in the C++ standard library.

16.4.6.12 Derived classes

An implementation may derive any class in the C++ standard library from a class with a name reserved to the implementation.

Certain classes defined in the C++ standard library are required to be derived from other classes in the C++ standard library. An implementation may derive such a class directly from the required base or indirectly through a hierarchy of base classes with names reserved to the implementation.

In any case:

(3.1) Every base class described as virtual shall be virtual;

(3.2) Every base class not specified as virtual shall not be virtual;

(3.3) Unless explicitly stated otherwise, types with distinct names shall be distinct types.\(^{183}\)

All types specified in the C++ standard library shall be non-final types unless otherwise specified.

16.4.6.13 Restrictions on exception handling

Any of the functions defined in the C++ standard library can report a failure by throwing an exception of a type described in its Throws: paragraph, or of a type derived from a type named in the Throws: paragraph that would be caught by an exception handler for the base type.

Functions from the C standard library shall not throw exceptions\(^{184}\) except when such a function calls a program-supplied function that throws an exception.\(^{185}\)

---

183 There is an implicit exception to this rule for types that are described as synonyms for basic integral types, such as size_t (17.2) and streamoff (29.5.2).

184 That is, the C library functions can all be treated as if they are marked noexcept. This allows implementations to make performance optimizations based on the absence of exceptions at runtime.

185 The functions qsort() and bsearch() (25.12) meet this condition.
Destructor operations defined in the C++ standard library shall not throw exceptions. Every destructor in the C++ standard library shall behave as if it had a non-throwing exception specification.

Functions defined in the C++ standard library that do not have a `Throws:` paragraph but do have a potentially-throwing exception specification may throw implementation-defined exceptions. Implementations should report errors by throwing exceptions of or derived from the standard exception classes (17.6.4.1, 17.9, 19.2).

An implementation may strengthen the exception specification for a non-virtual function by adding a non-throwing exception specification.

16.4.6.14 Restrictions on storage of pointers

Objects constructed by the standard library that may hold a user-supplied pointer value or an integer of type `std::intptr_t` shall store such values in a traceable pointer location (6.7.5.5.4).

16.4.6.15 Value of error codes

Certain functions in the C++ standard library report errors via a `std::error_code` (19.5.4.1) object. That object's `category()` member shall return `std::system_category()` for errors originating from the operating system, or a reference to an implementation-defined `error_category` object for errors originating elsewhere. The implementation shall define the possible values of `value()` for each of these error categories.

[Example 1: For operating systems that are based on POSIX, implementations should define the `std::system_category()` values as identical to the POSIX `errno` values, with additional values as defined by the operating system’s documentation. Implementations for operating systems that are not based on POSIX should define values identical to the operating system’s values. For errors that do not originate from the operating system, the implementation may provide enums for the associated values. — end example]

16.4.6.16 Moved-from state of library types

Objects of types defined in the C++ standard library may be moved from (11.4.5.3). Move operations may be explicitly specified or implicitly generated. Unless otherwise specified, such moved-from objects shall be placed in a valid but unspecified state.

---

186) In particular, they can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc` (17.6.4.1).
17  Language support library

17.1  General

This Clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 37.

Table 37: Language support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.2 Common definitions</td>
<td><code>&lt;cstdlib&gt;</code>, <code>&lt;cstdint&gt;</code></td>
</tr>
<tr>
<td>17.3 Implementation properties</td>
<td><code>&lt;cmath&gt;</code>, <code>&lt;climits&gt;</code>, <code>&lt;limits&gt;</code>, <code>&lt;version&gt;</code></td>
</tr>
<tr>
<td>17.4 Integer types</td>
<td><code>&lt;cstdint&gt;</code></td>
</tr>
<tr>
<td>17.5 Start and termination</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>17.6 Dynamic memory management</td>
<td><code>&lt;new&gt;</code></td>
</tr>
<tr>
<td>17.7 Type identification</td>
<td><code>&lt;typeinfo&gt;</code></td>
</tr>
<tr>
<td>17.8 Source location</td>
<td><code>&lt;source_location&gt;</code></td>
</tr>
<tr>
<td>17.9 Exception handling</td>
<td><code>&lt;exception&gt;</code></td>
</tr>
<tr>
<td>17.10 Initializer lists</td>
<td><code>&lt;initializer_list&gt;</code></td>
</tr>
<tr>
<td>17.11 Comparisons</td>
<td><code>&lt;compare&gt;</code></td>
</tr>
<tr>
<td>17.12 Coroutines</td>
<td><code>&lt;coroutine&gt;</code></td>
</tr>
<tr>
<td>17.13 Other runtime support</td>
<td><code>&lt;csetjmp&gt;</code>, <code>&lt;csignal&gt;</code>, <code>&lt;cstdarg&gt;</code>, <code>&lt;cstdlib&gt;</code></td>
</tr>
</tbody>
</table>

17.2  Common definitions

17.2.1  Header `<cstdlib>` synopsis

```cpp
namespace std {
    using ptrdiff_t = see below;
    using size_t = see below;
    using max_align_t = see below;
    using nullptr_t = decltype(nullptr);

    enum class byte : unsigned char {};

    // 17.2.5, byte type operations
    template<class IntType>
        constexpr byte& operator<<=(byte& b, IntType shift) noexcept;
    template<class IntType>
        constexpr byte operator<<(byte b, IntType shift) noexcept;
    template<class IntType>
        constexpr byte& operator>>=(byte& b, IntType shift) noexcept;
    template<class IntType>
        constexpr byte operator>>(byte b, IntType shift) noexcept;
    constexpr byte& operator|=(byte& l, byte r) noexcept;
    constexpr byte operator|(byte l, byte r) noexcept;
    constexpr byte& operator&=(byte& l, byte r) noexcept;
    constexpr byte operator&(byte l, byte r) noexcept;
    constexpr byte& operator^=(byte& l, byte r) noexcept;
    constexpr byte operator^(byte l, byte r) noexcept;
    constexpr byte operator~(byte b) noexcept;
```
template<class IntType>
    constexpr IntType to_integer(byte b) noexcept;

#define NULL see below
#define offsetof(P, D) see below

The contents and meaning of the header `<cstdlib>` are the same as the C standard library header `<stddef.h>`, except that it does not declare the type `wchar_t`, that it also declares the type `byte` and its associated operations (17.2.5), and as noted in 17.2.3 and 17.2.4.

See also: ISO C 7.19

17.2.2 Header `<cstdlib>` synopsis

```cpp
namespace std {
    using size_t = see below;
    using div_t = see below;
    using ldiv_t = see below;
    using lldiv_t = see below;
}
```

```cpp
#define NULL see below
#define EXIT_FAILURE see below
#define EXIT_SUCCESS see below
#define RAND_MAX see below
#define MB_CUR_MAX see below
```

namespace std {
    // Exposition-only function type aliases
    extern “C” using c_atexit_handler = void(); // exposition only
    extern “C++” using atexit_handler = void(); // exposition only
    extern “C” using c_compare_pred = int(const void*, const void*); // exposition only
    extern “C++” using compare_pred = int(const void*, const void*); // exposition only

    // 17.5, start and termination
    [[noreturn]] void abort() noexcept;
    int atexit(c_atexit_handler* func) noexcept;
    int atexit(atexit_handler* func) noexcept;
    int at_quick_exit(c_atexit_handler* func) noexcept;
    int at_quick_exit(atexit_handler* func) noexcept;
    [[noreturn]] void exit(int status);
    [[noreturn]] void _Exit(int status) noexcept;
    [[noreturn]] void quick_exit(int status) noexcept;
    char* getenv(const char* name);
    int system(const char* string);

    // 20.10.12, C library memory allocation
    void* aligned_alloc(size_t alignment, size_t size);
    void* calloc(size_t nmemb, size_t size);
    void free(void* ptr);
    void* malloc(size_t size);
    void* realloc(void* ptr, size_t size);

    double atof(const char* nptr);
    int atoi(const char* nptr);
    long atol(const char* nptr);
    long long atoll(const char* nptr);
    double strtod(const char* nptr, char** endptr);
    float strtof(const char* nptr, char** endptr);
    long double strtold(const char* nptr, char** endptr);
    long int strtol(const char* nptr, char** endptr, int base);
    long long int strtoll(const char* nptr, char** endptr, int base);
    unsigned long int strtoul(const char* nptr, char** endptr, int base);
```
1

The contents and meaning of the header `<cstdlib>` are the same as the C standard library header `<stdlib.h>`, except that it does not declare the type `wchar_t`, and except as noted in 17.2.3, 17.2.4, 17.5, 20.10.12, 21.5.6, 25.12, 26.6.10, and 26.8.2.

[Note 1: Several functions have additional overloads in this document, but they have the same behavior as in the C standard library (16.2). — end note]

See also: ISO C 7.22

17.2.3 Null pointers

The type `nullptr_t` is a synonym for the type of a `nullptr` expression, and it has the characteristics described in 6.8.2 and 7.3.12.

[Note 1: Although `nullptr`'s address cannot be taken, the address of another `nullptr_t` object that is an lvalue can be taken. — end note]

2 The macro `NULL` is an implementation-defined null pointer constant.\(^\text{187}\)

See also: ISO C 7.19

17.2.4 Sizes, alignments, and offsets

The macro `offsetof(type, member-designator)` has the same semantics as the corresponding macro in the C standard library header `<stddef.h>`, but accepts a restricted set of `type` arguments in this document. Use of the `offsetof` macro with a `type` other than a standard-layout class (11.2) is conditionally-supported.\(^\text{188}\)

\(^\text{187}\) Possible definitions include 0 and 0L, but not `(void*)0`.

\(^\text{188}\) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.
The expression `offsetof(type, member-designator)` is never type-dependent (13.8.3.3) and it is value-dependent (13.8.3.4) if and only if `type` is dependent. The result of applying the `offsetof` macro to a static data member or a function member is undefined. No operation invoked by the `offsetof` macro shall throw an exception and `noexcept(offsetof(type, member-designator))` shall be `true`.

The type `ptrdiff_t` is an implementation-defined signed integer type that can hold the difference of two subscripts in an array object, as described in 7.6.6.

The type `size_t` is an implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object (7.6.2.5).

[Note 1: It is recommended that implementations choose types for `ptrdiff_t` and `size_t` whose integer conversion ranks (6.8.5) are no greater than that of `signed long int` unless a larger size is necessary to contain all the possible values. —end note]

The type `max_align_t` is a trivial standard-layout type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context (6.7.6).

See also: ISO C 7.19

### 17.2.5 byte type operations

```cpp
template<class IntType>
constexpr byte& operator<<=(byte& b, IntType shift) noexcept;
    Constraints: `is_integral_v<IntType>` is true.
    Effects: Equivalent to: `return b = b << shift;`

template<class IntType>
constexpr byte operator<<(byte b, IntType shift) noexcept;
    Constraints: `is_integral_v<IntType>` is true.
    Effects: Equivalent to:
        `return static_cast<byte>(static_cast<unsigned int>(b) << shift);`

template<class IntType>
constexpr byte& operator>>=(byte& b, IntType shift) noexcept;
    Constraints: `is_integral_v<IntType>` is true.
    Effects: Equivalent to: `return b = b >> shift;`

template<class IntType>
constexpr byte operator>>(byte b, IntType shift) noexcept;
    Constraints: `is_integral_v<IntType>` is true.
    Effects: Equivalent to:
        `return static_cast<byte>(static_cast<unsigned int>(b) >> shift);`

template<class IntType>
constexpr byte& operator|=(byte& l, byte r) noexcept;
    Effects: Equivalent to: `return l = l | r;`

template<class IntType>
constexpr byte operator|(byte l, byte r) noexcept;
    Effects: Equivalent to:
        `return static_cast<byte>(static_cast<unsigned int>(l) | static_cast<unsigned int>(r));`

template<class IntType>
constexpr byte& operator&=(byte& l, byte r) noexcept;
    Effects: Equivalent to: `return l = l & r;`

template<class IntType>
constexpr byte operator&(byte l, byte r) noexcept;
    Effects: Equivalent to:
        `return static_cast<byte>(static_cast<unsigned int>(l) & static_cast<unsigned int>(r));`

template<class IntType>
constexpr byte& operator^=(byte& l, byte r) noexcept;
    Effects: Equivalent to: `return l = l ^ r;`
```

§ 17.2.5
constexpr byte operator^(byte l, byte r) noexcept;

**Effects**: Equivalent to:
\[
\text{return static_cast<\text{byte}> (\text{static_cast<\text{unsigned int}}(l) \ ^ \ \text{static_cast<\text{unsigned int}}(r));}
\]

constexpr byte operator~(byte b) noexcept;

**Effects**: Equivalent to:
\[
\text{return static_cast<\text{byte}> (~\text{static_cast<\text{unsigned int}}(b));}
\]

```
template<class IntType>
constexpr IntType to_integer(byte b) noexcept;
```

**Constraints**: `is_integral_v<IntType>` is true.

**Effects**: Equivalent to:
\[
\text{return static_cast<IntType> (b);}
\]

17.3 Implementation properties

### 17.3.1 General

The headers `<limits>` (17.3.3), `<climits>` (17.3.6), and `<cfloat>` (17.3.7) supply characteristics of implementation-dependent arithmetic types (6.8.2).

### 17.3.2 Header `<version>` synopsis

The header `<version>` supplies implementation-dependent information about the C++ standard library (e.g., version number and release date).

Each of the macros defined in `<version>` is also defined after inclusion of any member of the set of library headers indicated in the corresponding comment in this synopsis.

[Note 1: Future revisions of C++ might replace the values of these macros with greater values. — end note]

```c
#define __cpp_lib_addressof_constexpr 201603L // also in <memory>
#define __cpp_lib_allocator_traits_is_always_equal 201411L
// also in <memory>, <scoped_allocator>, <string>, <deque>, <forward_list>, <list>, <vector>,
// <map>, <set>, <unordered_map>, <unordered_set>
#define __cpp_lib_any 201606L // also in <any>
#define __cpp_lib_apply 201603L // also in <tuple>
#define __cpp_lib_array_constexpr 201811L // also in <iterator>, <array>
#define __cpp_lib_assume_aligned 201811L // also in <utility>
#define __cpp_lib_atomic_flag_test 201907L // also in <atomic>
#define __cpp_lib_atomic_float 201711L // also in <atomic>
#define __cpp_lib_atomic_is_always_lock_free 201603L // also in <atomic>
#define __cpp_lib_atomic_lock_free_type_aliases 201907L // also in <atomic>
#define __cpp_lib_atomic_shared_ptr 201711L // also in <memory>
#define __cpp_lib_atomic_value_initialization 201911L // also in <atomic>, <memory>
#define __cpp_lib_barrier 201907L // also in <memory>
#define __cpp_lib_bind_front 201907L // also in <functional>
#define __cpp_lib_complex_udls 201309L // also in <complex>
#define __cpp_lib_concepts 202002L // also in <concepts>
#define __cpp_lib_constexpr_algorithms 201806L // also in <algorithm>
#define __cpp_lib_constexpr_dynamic_alloc 201907L // also in <memory>
```
#define __cpp_lib_constexpr_functional 201907L // also in <functional>
#define __cpp_lib_constexpr_iterator 201811L // also in <iterator>
#define __cpp_lib_constexpr_memory 201811L // also in <memory>
#define __cpp_lib_constexpr_numeric 201911L // also in <numeric>
#define __cpp_lib_constexpr_string 201907L // also in <string>
#define __cpp_lib_constexpr_string_view 201811L // also in <string_view>
#define __cpp_lib_constexpr_tuple 201811L // also in <tuple>
#define __cpp_lib_constexpr_utility 201811L // also in <utility>
#define __cpp_lib_constexpr_vector 201907L // also in <vector>
#define __cpp_lib_coroutine 201911L // also in <coroutine>
#define __cpp_lib_coroutine 201911L // also in <coroutine>
#define __cpp_lib_coroutine 201806L // also in <new>
#define __cpp_lib_enable_shared_from_this 201603L // also in <memory>
#define __cpp_lib_erase_if 202002L // also in <string>, <deque>, <forward_list>, <list>, <vector>, <map>, <set>, <unordered_map>,
#define __cpp_lib_exchange_function 201304L // also in <utility>
#define __cpp_lib_execution 201902L // also in <execution>
#define __cpp_lib_filesytem 201703L // also in <filesystem>
#define __cpp_lib_format 201907L // also in <format>
#define __cpp_lib_format 201606L // also in <numeric>
#define __cpp_lib_generic_associative_lookup 201304L // also in <map>, <set>
#define __cpp_lib_generic_unordered_lookup 201811L // also in <unordered_map>, <unordered_set>
#define __cpp_lib_hardware_interference_size 201811L // also in <new>
#define __cpp_lib_hypot 201603L // also in <cmath>
#define __cpp_lib_hypot 201603L // also in <numeric>
#define __cpp_lib_incomplete_container_elements 201505L // also in <forward_list>, <list>, <vector>
#define __cpp_lib_int_pow2 202002L // also in <bit>
#define __cpp_lib_integer_comparison_functions 202002L // also in <utility>
#define __cpp_lib_integer_sequence 201304L // also in <numeric>
#define __cpp_lib_interface_types 201304L // also in <type_traits>
#define __cpp_lib_interpolate 201902L // also in <cmath>, <numeric>
#define __cpp_lib_invoke 201411L // also in <functional>
#define __cpp_lib_is_aggregate 201703L // also in <type_traits>
#define __cpp_lib_is_constant_evaluated 201811L // also in <type_traits>
#define __cpp_lib_is_final 201402L // also in <type_traits>
#define __cpp_lib_is_invocable 201703L // also in <type_traits>
#define __cpp_lib_is_layout_compatible 201907L // also in <type_traits>
#define __cpp_lib_is_notthrow_convertible 201806L // also in <type_traits>
#define __cpp_lib_is_pointer_interconvertible 201907L // also in <type_traits>
#define __cpp_lib_is_swapable 201806L // also in <type_traits>
#define __cpp_lib_jthread 201911L // also in <stop_token>, <thread>
#define __cpp_lib_latch 201907L // also in <latch>
#define __cpp_lib_launder 201606L // also in <new>
#define __cpp_lib_list_remove_return_type 201806L // also in <forward_list>, <list>
#define __cpp_lib_logical_traits 201510L // also in <type_traits>
#define __cpp_lib_make_from_tuple 201806L // also in <tuple>
#define __cpp_lib_make_reverse_iterator 201402L // also in <iterator>
#define __cpp_lib_make_unique 201304L // also in <iterator>
#define __cpp_lib_make_reverse_iterator 201402L // also in <iterator>
#define __cpp_lib_make_from_tuple 201606L // also in <functional>
#define __cpp_lib_make_unique 201304L // also in <iterator>
#define __cpp_lib_make_reverse_iterator 201402L // also in <iterator>
#define __cpp_lib_make_from_tuple 201606L // also in <optional>
#define __cpp_lib_nonmember_container_access 201411L // also in <array>, <deque>, <forward_list>, <iterator>, <list>, <map>, <regex>, <set>, <string>,
#define __cpp_lib_node_extract 201606L // also in <unordered_map>, <unordered_set>, <vector>
#define __cpp_lib_not_fcn 201603L // also in <functional>
#define __cpp_lib_nonmember_container_access 201411L // also in <array>, <deque>, <forward_list>, <iterator>, <list>, <map>, <regex>, <set>, <string>,
#define __cpp_lib_node_extract 201606L // also in <optional>
§ 17.3.3  Header `<limits>` synopsis

```cpp
namespace std {
    // 17.3.4, floating-point type properties
    enum float_round_style;
    enum float_denorm_style;

    // 17.3.5, class template numeric_limits
    template<class T> class numeric_limits;
    template<class T> class numeric_limits<const T>;
    template<class T> class numeric_limits<volatile T>;
    template<class T> class numeric_limits<const volatile T>;
    template<> class numeric_limits<bool>;
    template<> class numeric_limits<char>;
    template<> class numeric_limits<signed char>;
    template<> class numeric_limits<unsigned char>;
    template<> class numeric_limits<char8_t>;
    template<> class numeric_limits<char16_t>;
    template<> class numeric_limits<char32_t>;
    template<> class numeric_limits<wchar_t>;
}
```
17.3.4 Floating-point type properties

17.3.4.1 Type float_round_style

namespace std {
    enum float_round_style {
        round_indeterminate = -1,
        round_toward_zero = 0,
        round_to_nearest = 1,
        round_toward_infinity = 2,
        round_toward_neg_infinity = 3
    };
}

1 The rounding mode for floating-point arithmetic is characterized by the values:

1.1 — round_indeterminate if the rounding style is indeterminable
1.2 — round_toward_zero if the rounding style is toward zero
1.3 — round_to_nearest if the rounding style is to the nearest representable value
1.4 — round_toward_infinity if the rounding style is toward infinity
1.5 — round_toward_neg_infinity if the rounding style is toward negative infinity

17.3.4.2 Type float_denorm_style

namespace std {
    enum float_denorm_style {
        denorm_indeterminate = -1,
        denorm_absent = 0,
        denorm_present = 1
    };
}

1 The presence or absence of subnormal numbers (variable number of exponent bits) is characterized by the values:

1.1 — denorm_indeterminate if it cannot be determined whether or not the type allows subnormal values
1.2 — denorm_absent if the type does not allow subnormal values
1.3 — denorm_present if the type does allow subnormal values

17.3.5 Class template numeric_limits

17.3.5.1 General

The numeric_limits class template provides a C++ program with information about various properties of the implementation’s representation of the arithmetic types.

namespace std {
    template<class T> class numeric_limits {
        public:
            static constexpr bool is_specialized = false;
            static constexpr T min() noexcept { return T(); }
            static constexpr T max() noexcept { return T(); }
            static constexpr T lowest() noexcept { return T(); }
    };
}

§ 17.3.5.1 513
For all members declared `static constexpr` in the `numeric_limits` template, specializations shall define these values in such a way that they are usable as constant expressions.

For the `numeric_limits` primary template, all data members are value-initialized and all member functions return a value-initialized object.

[Note 1: This means all members have zero or `false` values unless `numeric_limits` is specialized for a type. — end note]

Specializations shall be provided for each arithmetic type, both floating-point and integer, including `bool`. The member `is_specialized` shall be `true` for all such specializations of `numeric_limits`.

The value of each member of a specialization of `numeric_limits` on a cv-qualified type `cv T` shall be equal to the value of the corresponding member of the specialization on the unqualified type `T`.

Non-arithmetic standard types, such as `complex<T>` (26.4.3), shall not have specializations.

### 17.3.5.2 numeric_limits members

Each member function defined in this subclause is signal-safe (17.13.5).

**static constexpr T min() noexcept;**

Minimum finite value.\(^{189}\)

For floating-point types with subnormal numbers, returns the minimum positive normalized value.

Meaningful for all specializations in which `is_bounded` != `false`, or `is_bounded == false && is_singed` = `false`.

---

\(^{189}\) Equivalent to `CHAR_MIN`, `SHRT_MIN`, `FLT_MIN`, `DBL_MIN`, etc.
static constexpr T max() noexcept;
      Maximum finite value.\(^{190}\)
      Meaningful for all specializations in which is_bounded != false.

static constexpr T lowest() noexcept;
      A finite value x such that there is no other finite value y where y < x.\(^{191}\)
      Meaningful for all specializations in which is_bounded != false.

static constexpr int digits;
      Number of radix digits that can be represented without change.
      For integer types, the number of non-sign bits in the representation.
      For floating-point types, the number of radix digits in the mantissa.\(^{192}\)

static constexpr int digits10;
      Number of base 10 digits that can be represented without change.\(^{193}\)
      Meaningful for all specializations in which is_bounded != false.

static constexpr int max_digits10;
      Number of base 10 digits required to ensure that values which differ are always differentiated.
      Meaningful for all floating-point types.

static constexpr bool is_signed;
      true if the type is signed.
      Meaningful for all specializations.

static constexpr bool is_integer;
      true if the type is integer.
      Meaningful for all specializations.

static constexpr bool is_exact;
      true if the type uses an exact representation. All integer types are exact, but not all exact types are integer.
      For example, rational and fixed-exponent representations are exact but not integer.
      Meaningful for all specializations.

static constexpr int radix;
      For floating-point types, specifies the base or radix of the exponent representation (often 2).\(^{194}\)
      For integer types, specifies the base of the representation.\(^{195}\)
      Meaningful for all specializations.

static constexpr T epsilon() noexcept;
      Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.\(^{196}\)
      Meaningful for all floating-point types.

static constexpr T round_error() noexcept;
      Measure of the maximum rounding error.\(^{197}\)

\(^{190}\) Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.
\(^{191}\) lowest() is necessary because not all floating-point representations have a smallest (most negative) value that is the negative of the largest (most positive) finite value.
\(^{192}\) Equivalent to FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG.
\(^{193}\) Equivalent to FLT_DIG, DBL_DIG, LDBL_DIG.
\(^{194}\) Equivalent to FLT_RADIX.
\(^{195}\) Distinguishes types with bases other than 2 (e.g. BCD).
\(^{196}\) Equivalent to FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON.
\(^{197}\) Rounding error is described in LIA-1 Section 5.2.4 and Annex C Rationale Section C.5.2.4 — Rounding and rounding constants.
static constexpr int min_exponent;

Minimum negative integer such that \textit{radix} raised to the power of one less than that integer is a normalized floating-point number.\footnote{Equivalent to FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP.}

Meaningful for all floating-point types.

static constexpr int min_exponent10;

Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.\footnote{Equivalent to FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP.}

Meaningful for all floating-point types.

static constexpr int max_exponent;

Maximum positive integer such that \textit{radix} raised to the power one less than that integer is a representable finite floating-point number.\footnote{Equivalent to FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.}

Meaningful for all floating-point types.

static constexpr int max_exponent10;

Maximum positive integer such that 10 raised to that power is in the range of representable finite floating-point numbers.\footnote{Equivalent to FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP.}

Meaningful for all floating-point types.

static constexpr bool has_infinity;

\texttt{true} if the type has a representation for positive infinity.

Meaningful for all floating-point types.

\textbf{Shall be} \texttt{true} for all specializations in which \textit{is_iec559} \footnote{Required by LIA-1.} \neq \texttt{false}.

static constexpr bool has_quiet_NaN;

\texttt{true} if the type has a representation for a quiet (non-signaling) “Not a Number.”\footnote{Required by LIA-1.}

Meaningful for all floating-point types.

\textbf{Shall be} \texttt{true} for all specializations in which \textit{is_iec559} \footnote{Required by LIA-1.} \neq \texttt{false}.

static constexpr bool has_signaling_NaN;

\texttt{true} if the type has a representation for a signaling “Not a Number”.\footnote{Required by LIA-1.}

Meaningful for all floating-point types.

\textbf{Shall be} \texttt{true} for all specializations in which \textit{is_iec559} \footnote{Required by LIA-1.} \neq \texttt{false}.

static constexpr float_denorm_style has_denorm;

\texttt{denorm_present} if the type allows subnormal values (variable number of exponent bits)\footnote{Required by LIA-1.}, \texttt{denorm_-absent} if the type does not allow subnormal values, and \texttt{denorm_indeterminate} if it is indeterminate at compile time whether the type allows subnormal values.

Meaningful for all floating-point types.

\textbf{static constexpr bool has_denorm_loss;}

\texttt{true} if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.\footnote{See ISO/IEC/IEEE 60559.}

\textbf{static constexpr T infinity() noexcept;}

Representation of positive infinity, if available.\footnote{Required by LIA-1.}
Meaningful for all specializations for which $\text{has\_infinity} \neq \text{false}$. Required in specializations for which $\text{is\_iec559} \neq \text{false}$.

```cpp
static constexpr T quiet_NaN() noexcept;
```

Representation of a quiet “Not a Number”, if available.\(^{207}\)

Meaningful for all specializations for which $\text{has\_quiet\_NaN} \neq \text{false}$. Required in specializations for which $\text{is\_iec559} \neq \text{false}$.

```cpp
static constexpr T signaling_NaN() noexcept;
```

Representation of a signaling “Not a Number”, if available.\(^{208}\)

Meaningful for all specializations for which $\text{has\_signaling\_NaN} \neq \text{false}$. Required in specializations for which $\text{is\_iec559} \neq \text{false}$.

```cpp
static constexpr T denorm_min() noexcept;
```

Minimum positive subnormal value.\(^{209}\)

Meaningful for all floating-point types.

In specializations for which $\text{has\_denorm} == \text{false}$, returns the minimum positive normalized value.

```cpp
static constexpr bool is_iec559;
```

true if and only if the type adheres to ISO/IEC/IEEE 60559.\(^{210}\)

Meaningful for all floating-point types.

```cpp
static constexpr bool is_bounded;
```

true if the set of values representable by the type is finite.\(^{211}\)

[Note 1: All fundamental types (6.8.2) are bounded. This member would be false for arbitrary precision types. — end note]

Meaningful for all specializations.

```cpp
static constexpr bool is_modulo;
```

true if the type is modulo.\(^{212}\) A type is modulo if, for any operation involving $+$, $-$, or $\ast$ on values of that type whose result would fall outside the range $[\text{min()}, \text{max()}]$, the value returned differs from the true value by an integer multiple of $\text{max()} - \text{min()} + 1$.

[Example 1: is\_modulo is false for signed integer types (6.8.2) unless an implementation, as an extension to this document, defines signed integer overflow to wrap. — end example]

Meaningful for all specializations.

```cpp
static constexpr bool traps;
```

true if, at the start of the program, there exists a value of the type that would cause an arithmetic operation using that value to trap.\(^{213}\)

Meaningful for all specializations.

```cpp
static constexpr bool tinyness_before;
```

true if tinyness is detected before rounding.\(^ {214}\)

Meaningful for all floating-point types.

```cpp
static constexpr float_round_style round_style;
```

The rounding style for the type.\(^ {215}\)

\(^{207}\) Required by LIA-1.

\(^{208}\) Required by LIA-1.

\(^{209}\) Required by LIA-1.


\(^{211}\) Required by LIA-1.

\(^{212}\) Required by LIA-1.

\(^{213}\) Required by LIA-1.

\(^{214}\) Refer to ISO/IEC/IEEE 60559. Required by LIA-1.

\(^{215}\) Equivalent to FLT\_ROUNDS. Required by LIA-1.
Meaningful for all floating-point types. Specializations for integer types shall return \texttt{round\textunderscore toward\textunderscore zero}.

17.3.5.3 \texttt{numeric\_limits} specializations

All members shall be provided for all specializations. However, many values are only required to be meaningful under certain conditions (for example, \texttt{epsilon()} is only meaningful if \texttt{is\_integer} is \texttt{false}). Any value that is not “meaningful” shall be set to 0 or \texttt{false}.

\begin{verbatim}
namespace std {
    template<> class numeric_limits<float> {
        public:
            static constexpr bool is_specialized = true;
            static constexpr float min() noexcept { return 1.17549435E-38F; }
            static constexpr float max() noexcept { return 3.40282347E+38F; }
            static constexpr float lowest() noexcept { return -3.40282347E+38F; }
            static constexpr int digits = 24;
            static constexpr int digits10 = 6;
            static constexpr int max_digits10 = 9;
            static constexpr bool is_signed = true;
            static constexpr bool is_integer = false;
            static constexpr bool is_exact = false;
            static constexpr int radix = 2;
            static constexpr float epsilon() noexcept { return 1.19209290E-07F; }
            static constexpr float round_error() noexcept { return 0.5F; }
            static constexpr int min_exponent = -125;
            static constexpr int min_exponent10 = -37;
            static constexpr int max_exponent = +128;
            static constexpr int max_exponent10 = +38;
            static constexpr bool has_infinity = true;
            static constexpr bool has_quiet_NaN = true;
            static constexpr bool has_signaling_NaN = true;
            static constexpr float_denorm_style has_denorm = denorm_absent;
            static constexpr bool has_denorm_loss = false;
            static constexpr float infinity() noexcept { return value; }
            static constexpr float quiet_NaN() noexcept { return value; }
            static constexpr float signaling_NaN() noexcept { return value; }
            static constexpr float denorm_min() noexcept { return min(); }
            static constexpr bool is_iec559 = true;
            static constexpr bool is_bounded = true;
            static constexpr bool is_modulo = false;
            static constexpr bool traps = true;
            static constexpr bool tinyness_before = true;
            static constexpr float_round_style round_style = round_to_nearest;
        }
    }
}\end{verbatim}

The specialization for \texttt{bool} shall be provided as follows:

\begin{verbatim}
namespace std {
    template<> class numeric_limits<bool> {
        public:
            static constexpr bool is_specialized = true;
            static constexpr bool min() noexcept { return false; }
        }
}\end{verbatim}
static constexpr bool max() noexcept { return true; }
static constexpr bool lowest() noexcept { return false; }

static constexpr int digits = 1;
static constexpr int digits10 = 0;
static constexpr int max_digits10 = 0;

static constexpr bool is_signed = false;
static constexpr bool is_integer = true;
static constexpr bool is_exact = true;
static constexpr int radix = 2;
static constexpr bool epsilon() noexcept { return 0; }
static constexpr bool round_error() noexcept { return 0; }

static constexpr int min_exponent = 0;
static constexpr int min_exponent10 = 0;
static constexpr int max_exponent = 0;
static constexpr int max_exponent10 = 0;

static constexpr bool has_infinity = false;
static constexpr bool has_quiet_NaN = false;
static constexpr bool has_signaling_NaN = false;
static constexpr float_denorm_style has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;
static constexpr bool infinity() noexcept { return 0; }
static constexpr bool quiet_NaN() noexcept { return 0; }
static constexpr bool signaling_NaN() noexcept { return 0; }
static constexpr bool denorm_min() noexcept { return 0; }

static constexpr bool is_iec559 = false;
static constexpr bool is_bounded = true;
static constexpr bool is_modulo = false;
static constexpr bool traps = false;
static constexpr bool tinyness_before = false;
static constexpr float_round_style round_style = round_toward_zero;
};

17.3.6 Header <climits> synopsis

#define CHAR_BIT see below
#define SCHAR_MIN see below
#define SCHAR_MAX see below
#define UCHAR_MAX see below
#define CHAR_MIN see below
#define CHAR_MAX see below
#define MB_LEN_MAX see below
#define CHAR_MAX see below
#define SHRT_MIN see below
#define SHRT_MAX see below
#define USHRT_MAX see below
#define INT_MIN see below
#define INT_MAX see below
#define UINT_MAX see below
#define LONG_MIN see below
#define LONG_MAX see below
#define ULONG_MAX see below
#define LLONG_MIN see below
#define LLONG_MAX see below
#define ULLONG_MAX see below

1 The header <climits> defines all macros the same as the C standard library header <limits.h>.

>Note 1: The types of the constants defined by macros in <climits> are not required to match the types to which the macros refer. — end note]
17.3.7 Header `<cfloat>` synopsis [cfloat.syn]

#define FLT_ROUNDS see below
#define FLT_EVAL_METHOD see below
#define FLT_HAS_SUBNORM see below
#define DBL_HAS_SUBNORM see below
#define LDBL_HAS_SUBNORM see below
#define FLT_RADIX see below
#define FLT_MANT_DIG see below
#define DBL_MANT_DIG see below
#define LDBL_MANT_DIG see below
#define FLT_DECIMAL_DIG see below
#define DECIMAL_DIG see below
#define FLT_DIG see below
#define DBL_DIG see below
#define LDBL_DIG see below
#define FLT_MIN_EXP see below
#define DBL_MIN_EXP see below
#define LDBL_MIN_EXP see below
#define FLT_MIN_10_EXP see below
#define DBL_MIN_10_EXP see below
#define LDBL_MIN_10_EXP see below
#define FLT_MAX_EXP see below
#define DBL_MAX_EXP see below
#define LDBL_MAX_EXP see below
#define FLT_MAX see below
#define DBL_MAX see below
#define LDBL_MAX see below
#define FLT_TRUE_MIN see below
#define DBL_TRUE_MIN see below
#define LDBL_TRUE_MIN see below
#define FLT_EPSILON see below
#define DBL_EPSILON see below
#define LDBL_EPSILON see below
#define FLT_MIN see below
#define DBL_MIN see below
#define LDBL_MIN see below
#define FLT_MIN_10 see below
#define DBL_MIN_10 see below
#define LDBL_MIN_10 see below
#define FLT_MAX see below
#define DBL_MAX see below
#define LDBL_MAX see below
#define FLT_EPSILON see below
#define DBL_EPSILON see below
#define LDBL_EPSILON see below
#define FLT_MIN see below
#define DBL_MIN see below
#define LDBL_MIN see below
#define FLT_TRUE_MIN see below
#define DBL_TRUE_MIN see below
#define LDBL_TRUE_MIN see below
#define DECIMAL_DIG see below
#define FLT_DIG see below
#define DBL_DIG see below
#define LDBL_DIG see below
#define FLT_MIN see below
#define DBL_MIN see below
#define LDBL_MIN see below
#define FLT_MAX see below
#define DBL_MAX see below
#define LDBL_MAX see below
#define FLT_EPSILON see below
#define DBL_EPSILON see below
#define LDBL_EPSILON see below
#define FLT_MIN see below
#define DBL_MIN see below
#define LDBL_MIN see below
#define FLT_TRUE_MIN see below
#define DBL_TRUE_MIN see below
#define LDBL_TRUE_MIN see below

1 The header `<cfloat>` defines all macros the same as the C standard library header `<float.h>`.

See also: ISO C 5.2.4.2.2

17.4 Integer types [cstdint]

17.4.1 General [cstdint.general]

1 The header `<cstdint>` (17.4.2) supplies integer types having specified widths, and macros that specify limits of integer types.

17.4.2 Header `<cstdint>` synopsis [cstdint.syn]

namespace std {
    using int8_t = signed integer type; // optional
    using int16_t = signed integer type; // optional
    using int32_t = signed integer type; // optional
    using int64_t = signed integer type; // optional
    using int_fast8_t = signed integer type;
    using int_fast16_t = signed integer type;
    using int_fast32_t = signed integer type;
}

§ 17.4.2 520
using int_fast64_t = signed integer type;
using int_least8_t = signed integer type;
using int_least16_t = signed integer type;
using int_least32_t = signed integer type;
using int_least64_t = signed integer type;
using intmax_t = signed integer type;
using intptr_t = signed integer type; // optional
using uint8_t = unsigned integer type; // optional
using uint16_t = unsigned integer type; // optional
using uint32_t = unsigned integer type; // optional
using uint64_t = unsigned integer type; // optional
using uint_fast8_t = unsigned integer type;
using uint_fast16_t = unsigned integer type;
using uint_fast32_t = unsigned integer type;
using uint_fast64_t = unsigned integer type;
using uint_least8_t = unsigned integer type;
using uint_least16_t = unsigned integer type;
using uint_least32_t = unsigned integer type;
using uint_least64_t = unsigned integer type;
using uintmax_t = unsigned integer type;
using uintptr_t = unsigned integer type; // optional

1 The header also defines numerous macros of the form:

\[
\text{INT\{FAST LEAST\}{8 16 32 64}\_MIN}
\]

\[
\text{[U]INT\{FAST LEAST\}{8 16 32 64}\_MAX}
\]

\[
\text{INT\{MAX PTR\}\_MIN}
\]

\[
\text{[U]INT\{MAX PTR\}\_MAX}
\]

\[
\{\text{PTRDIFF SIG_ATOMIC WCHAR WINT}\}{\_MAX \_MIN}
\]

\[
\text{SIZE\_MAX}
\]

plus function macros of the form:

\[
\text{[U]INT\{8 16 32 64 MAX\}\_C}
\]

2 The header defines all types and macros the same as the C standard library header \texttt{<stdint.h>}. See also: ISO C 7.20

17.5 Startup and termination

[ support.start.term ]

1 \[ Note 1: \) The header \texttt{<cstdlib> (17.2.2) declares the functions described in this subclause. \end note ]

[[noreturn]] void _Exit(int status) noexcept;

2 \[ Effects: \) This function has the semantics specified in the C standard library.

3 \[ Remarks: \) The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to \texttt{atexit()} (6.9.3.4). The function \_Exit is signal-safe (17.13.5).

[[noreturn]] void abort() noexcept;

4 \[ Effects: \) This function has the semantics specified in the C standard library.

5 \[ Remarks: \) The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to \texttt{atexit()} (6.9.3.4). The function \texttt{abort} is signal-safe (17.13.5).

int atexit(c-\texttt{atexit-handler}* f) noexcept;
int atexit(atexit-handler* f) noexcept;

Effects: The atexit() functions register the function pointed to by f to be called without arguments at normal program termination. It is unspecified whether a call to atexit() that does not happen before (6.9.2) a call to exit() will succeed.

[Note 2: The atexit() functions do not introduce a data race (16.4.6.10). — end note]

Implementation limits: The implementation shall support the registration of at least 32 functions.

Returns: The atexit() function returns zero if the registration succeeds, nonzero if it fails.

[[noreturn]] void exit(int status);

Effects:

(9.1) First, objects with thread storage duration and associated with the current thread are destroyed. Next, objects with static storage duration are destroyed and functions registered by calling atexit are called. See 6.9.3.4 for the order of destructions and calls. (Objects with automatic storage duration are not destroyed as a result of calling exit().)\[216\]

If control leaves a registered function called by exit because the function does not provide a handler for a thrown exception, the function std::terminate shall be called (14.6.2).

(9.2) Next, all open C streams (as mediated by the function signatures declared in <cstdio> (29.12.1)) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling tmpfile() are removed.

(9.3) Finally, control is returned to the host environment. If status is zero or EXIT_SUCCESS, an implementation-defined form of the status successful termination is returned. If status is EXIT_FAILURE, an implementation-defined form of the status unsuccessful termination is returned. Otherwise the status returned is implementation-defined.\[218\]

int at_quick_exit(c-atexit-handler* f) noexcept;
int at_quick_exit(atexit-handler* f) noexcept;

Effects: The at_quick_exit() functions register the function pointed to by f to be called without arguments when quick_exit is called. It is unspecified whether a call to at_quick_exit() that does not happen before (6.9.2) all calls to quick_exit will succeed.

[Note 3: The at_quick_exit() functions do not introduce a data race (16.4.6.10). — end note]

[Note 4: The order of registration could be indeterminate if at_quick_exit was called from more than one thread. — end note]

[Note 5: The at_quick_exit registrations are distinct from the atexit registrations, and applications might need to call both registration functions with the same argument. — end note]

Implementation limits: The implementation shall support the registration of at least 32 functions.

Returns: Zero if the registration succeeds, nonzero if it fails.

[[noreturn]] void quick_exit(int status) noexcept;

Effects: Functions registered by calls to at_quick_exit are called in the reverse order of their registration, except that a function shall be called after any previously registered functions that had already been called at the time it was registered. Objects shall not be destroyed as a result of calling quick_exit. If control leaves a registered function called by quick_exit because the function does not provide a handler for a thrown exception, the function std::terminate shall be called.

[Note 6: A function registered via at_quick_exit is invoked by the thread that calls quick_exit, which can be a different thread than the one that registered it, so registered functions cannot rely on the identity of objects with thread storage duration. — end note]

After calling registered functions, quick_exit shall call _Exit(status).

Remarks: The function quick_exit is signal-safe (17.13.5) when the functions registered with at_-quick_exit are.

\[216\] A function is called for every time it is registered.
\[217\] Objects with automatic storage duration are all destroyed in a program whose main function (6.9.3.1) contains no objects with automatic storage duration and executes the call to exit(). Control can be transferred directly to such a main function by throwing an exception that is caught in main.
\[218\] The macros EXIT_FAILURE and EXIT_SUCCESS are defined in <cstdlib> (17.2.2).
17.6 Dynamic memory management 
17.6.1 General

The header `<new>` defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

17.6.2 Header `<new>` synopsis

```cpp
namespace std {
    // 17.6.4, storage allocation errors
class bad_alloc;
class bad_array_new_length;

    struct destroying_delete_t {
        explicit destroying_delete_t() = default;
    };
    inline constexpr destroying_delete_t destroying_delete{};

    // global operator new control
    enum class align_val_t : size_t {
    }

    struct noexcept_t { explicit noexcept_t() = default; }
    extern const noexcept_t noexcept;

    using new_handler = void (*)();
    new_handler get_new_handler() noexcept;
    new_handler set_new_handler(new_handler new_p) noexcept;

    // 17.6.5, pointer optimization barrier
    template<class T> [[nodiscard]] constexpr T* launder(T* p) noexcept;

    // 17.6.6, hardware interference size
    inline constexpr size_t hardware_destructive_interference_size = implementation-defined;
    inline constexpr size_t hardware_constructive_interference_size = implementation-defined;
}

// 17.6.3, storage allocation and deallocation
[[nodiscard]] void* operator new(std::size_t size);
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment);
[[nodiscard]] void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment, const std::nothrow_t&) noexcept;
void operator delete(void* ptr) noexcept;
void operator delete(void* ptr, std::size_t size) noexcept;
void operator delete(void* ptr, std::align_val_t alignment) noexcept;
void operator delete(void* ptr, std::size_t size, std::align_val_t alignment) noexcept;
void operator delete(void* ptr, const std::nothrow_t&) noexcept;
void operator delete(void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;

[[nodiscard]] void* operator new[](std::size_t size);
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment);
[[nodiscard]] void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment, const std::nothrow_t&) noexcept;
void operator delete[](void* ptr) noexcept;
void operator delete[](void* ptr, std::size_t size) noexcept;
void operator delete[](void* ptr, std::align_val_t alignment) noexcept;
void operator delete[](void* ptr, std::size_t size, std::align_val_t alignment) noexcept;
void operator delete[](void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;
```
[new.delete]

17.6.3 Storage allocation and deallocation

17.6.3.1 General

Except where otherwise specified, the provisions of 6.7.5.5 apply to the library versions of \texttt{operator new} and \texttt{operator delete}. If the value of an alignment argument passed to any of these functions is not a valid alignment value, the behavior is undefined.

17.6.3.2 Single-object forms

Effects: The allocation functions (6.7.5.5.2) called by a \texttt{new-expression} (7.6.2.8) to allocate \texttt{size} bytes of storage. The second form is called for a type with new-extended alignment, and the first form is called otherwise.

Replaceable: A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

Required behavior: Return a non-null pointer to suitably aligned storage (6.7.5.5), or else throw a \texttt{bad_alloc} exception. This requirement is binding on any replacement versions of these functions.

Default behavior:

1. Executes a loop: Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the C standard library functions \texttt{malloc} or \texttt{aligned_alloc} is unspecified.
2. Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the current \texttt{new_handler} (17.6.4.5) is a null pointer value, throws \texttt{bad_alloc}.
3. Otherwise, the function calls the current \texttt{new_handler} function (17.6.4.3). If the called function returns, the loop repeats.
4. The loop terminates when an attempt to allocate the requested storage is successful or when a called \texttt{new_handler} function does not return.

Effects: Same as above, except that these are called by a placement version of a \texttt{new-expression} when a C++ program prefers a null pointer result as an error indication, instead of a \texttt{bad_alloc} exception.

Replaceable: A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

Required behavior: Return a non-null pointer to suitably aligned storage (6.7.5.5), or else return a null pointer. Each of these nothrow versions of \texttt{operator new} returns a pointer obtained as if acquired from the (possibly replaced) corresponding non-placement function. This requirement is binding on any replacement versions of these functions.

Default behavior: Calls \texttt{operator new(size)}, or \texttt{operator new(size, alignment)}, respectively. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

Example 1:

```cpp
T* p1 = new T; // throws bad_alloc if it fails
T* p2 = new(nothrow) T; // returns nullptr if it fails
```

---end example---

§ 17.6.3.2
void operator delete(void* ptr, std::size_t size, std::align_val_t alignment) noexcept;

Preconditions: ptr is a null pointer or its value represents the address of a block of memory allocated by an earlier call to a (possibly replaced) operator new(std::size_t) or operator new(std::size_t, std::align_val_t) which has not been invalidated by an intervening call to operator delete.

If an implementation has strict pointer safety (6.7.5.5.4) then ptr is a safely-derived pointer.

If the alignment parameter is not present, ptr was returned by an allocation function without an alignment parameter. If present, the alignment argument is equal to the alignment argument passed to the allocation function that returned ptr. If present, the size argument is equal to the size argument passed to the allocation function that returned ptr.

Effects: The deallocation functions (6.7.5.5.3) called by a delete-expression (7.6.2.9) to render the value of ptr invalid.

Replaceable: A C++ program may define functions with any of these function signatures, and thereby displace the default versions defined by the C++ standard library. If a function without a size parameter is defined, the program should also define the corresponding function with a size parameter. If a function with a size parameter is defined, the program shall also define the corresponding version without the size parameter.

[Note 1: The default behavior below might change in the future, which will require replacing both deallocation functions when replacing the allocation function. — end note]

Required behavior: A call to an operator delete with a size parameter may be changed to a call to the corresponding operator delete without a size parameter, without affecting memory allocation.

[Note 2: A conforming implementation is for operator delete(void* ptr, std::size_t size) to simply call operator delete(ptr). — end note]

Default behavior: The functions that have a size parameter forward their other parameters to the corresponding function without a size parameter.

[Note 3: See the note in the above Replaceable: paragraph. — end note]

Default behavior: If ptr is null, does nothing. Otherwise, reclaims the storage allocated by the earlier call to operator new.

Remarks: It is unspecified under what conditions part or all of such reclaimed storage will be allocated by subsequent calls to operator new or any of aligned_alloc, calloc, malloc, or realloc, declared in <cstdlib> (17.2.2).

void operator delete(void* ptr, const std::nothrow_t&) noexcept;
void operator delete(void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;

Preconditions: ptr is a null pointer or its value represents the address of a block of memory allocated by an earlier call to a (possibly replaced) operator new(std::size_t) or operator new(std::size_t, std::align_val_t) which has not been invalidated by an intervening call to operator delete.

If an implementation has strict pointer safety (6.7.5.5.4) then ptr is a safely-derived pointer.

If the alignment parameter is not present, ptr was returned by an allocation function without an alignment parameter. If present, the alignment argument is equal to the alignment argument passed to the allocation function that returned ptr.

Effects: The deallocation functions (6.7.5.5.3) called by the implementation to render the value of ptr invalid when the constructor invoked from a nothrow placement version of the new-expression throws an exception.

Replaceable: A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

Default behavior: Calls operator delete(ptr), or operator delete(ptr, alignment), respectively.

17.6.3.3 Array forms

[[nodiscard]] void* operator new[](std::size_t size);

§ 17.6.3.3 525
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment);

Effects: The allocation functions (6.7.5.5.2) called by the array form of a new-expression (7.6.2.8) to allocate size bytes of storage. The second form is called for a type with new-extended alignment, and the first form is called otherwise.\(^{219}\)

Replaceable: A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

Required behavior: Same as for the corresponding single-object forms. This requirement is binding on any replacement versions of these functions.

Default behavior: Returns operator new(size), or operator new(size, alignment), respectively.

[[nodiscard]] void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment, const std::nothrow_t&) noexcept;

Effects: Same as above, except that these are called by a placement version of a new-expression when a C++ program prefers a null pointer result as an error indication, instead of a bad_alloc exception.

Replaceable: A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

Required behavior: Return a non-null pointer to suitably aligned storage (6.7.5.5), or else return a null pointer. Each of these nothrow versions of operator new[] returns a pointer obtained as if acquired from the (possibly replaced) corresponding non-placement function. This requirement is binding on any replacement versions of these functions.

Default behavior: Calls operator new[](size), or operator new[](size, alignment), respectively. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

void operator delete[](void* ptr) noexcept;
void operator delete[](void* ptr, std::size_t size) noexcept;
void operator delete[](void* ptr, std::align_val_t alignment) noexcept;
void operator delete[](void* ptr, std::size_t size, std::align_val_t alignment) noexcept;

Preconditions: ptr is a null pointer or its value represents the address of a block of memory allocated by an earlier call to a (possibly replaced) operator new[](std::size_t) or operator new[](std::size_t, std::align_val_t) which has not been invalidated by an intervening call to operator delete[].

If an implementation has strict pointer safety (6.7.5.5.4) then ptr is a safely-derived pointer.

If the alignment parameter is not present, ptr was returned by an allocation function without an alignment parameter. If present, the alignment argument is equal to the alignment argument passed to the allocation function that returned ptr. If present, the size argument is equal to the size argument passed to the allocation function that returned ptr.

Effects: The deallocation functions (6.7.5.5.3) called by the array form of a delete-expression to render the value of ptr invalid.

Replaceable: A C++ program may define functions with any of these function signatures, and thereby displace the default versions defined by the C++ standard library. If a function without a size parameter is defined, the program should also define the corresponding function with a size parameter. If a function with a size parameter is defined, the program shall also define the corresponding version without the size parameter.

[Note 1: The default behavior below might change in the future, which will require replacing both deallocation functions when replacing the allocation function. — end note]

Required behavior: A call to an operator delete[] with a size parameter may be changed to a call to the corresponding operator delete[] without a size parameter, without affecting memory allocation.

[Note 2: A conforming implementation is for operator delete[](void* ptr, std::size_t size) to simply call operator delete[](ptr). — end note]

\(^{219}\) It is not the direct responsibility of operator new[] or operator delete[] to note the repetition count or element size of the array. Those operations are performed elsewhere in the array new and delete expressions. The array new expression, can, however, increase the size argument to operator new[] to obtain space to store supplemental information.
Default behavior: The functions that have a size parameter forward their other parameters to the corresponding function without a size parameter. The functions that do not have a size parameter forward their parameters to the corresponding operator delete (single-object) function.

```c++
void operator delete[](void* ptr, const std::nothrow_t&) noexcept;
void operator delete[](void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;
```

Preconditions: `ptr` is a null pointer or its value represents the address of a block of memory allocated by an earlier call to a (possibly replaced) `operator new[](std::size_t)` or `operator new[](std::size_t, std::align_val_t)` which has not been invalidated by an intervening call to `operator delete[]`.

If an implementation has strict pointer safety (6.7.5.5.4) then `ptr` is a safely-derived pointer.

If the alignment parameter is not present, `ptr` was returned by an allocation function without an alignment parameter. If present, the alignment argument is equal to the alignment argument passed to the allocation function that returned `ptr`.

Effects: The deallocation functions (6.7.5.5.3) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a nothrow placement version of the array new-expression throws an exception.

Replaceable: A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

Default behavior: Calls `operator delete[](ptr)`, or `operator delete[](ptr, alignment)`, respectively.

### 17.6.3.4 Non-allocating forms [new.delete.placement]

These functions are reserved; a C++ program may not define functions that displace the versions in the C++ standard library (16.4.5). The provisions of 6.7.5.5 do not apply to these reserved placement forms of `operator new` and `operator delete`.

```c++
[[nodiscard]] void* operator new(std::size_t size, void* ptr) noexcept;
```

Returns: `ptr`.

Remarks: Intentionally performs no other action.

[Example 1: This can be useful for constructing an object at a known address:
```c++
void* place = operator new(sizeof(Something));
Something* p = new (place) Something();
```
—end example]

```c++
[[nodiscard]] void* operator new[](std::size_t size, void* ptr) noexcept;
```

Returns: `ptr`.

Remarks: Intentionally performs no other action.

```c++
void operator delete(void* ptr, void*) noexcept;
```

Preconditions: If an implementation has strict pointer safety (6.7.5.5.4) then `ptr` is a safely-derived pointer.

Effects: Intentionally performs no action.

Remarks: Default function called when any part of the initialization in a placement new-expression that invokes the library’s non-array placement operator new terminates by throwing an exception (7.6.2.8).

```c++
void operator delete[](void* ptr, void*) noexcept;
```

Preconditions: If an implementation has strict pointer safety (6.7.5.5.4) then `ptr` is a safely-derived pointer.

Effects: Intentionally performs no action.

Remarks: Default function called when any part of the initialization in a placement new-expression that invokes the library’s array placement operator new terminates by throwing an exception (7.6.2.8).
17.6.3.5 Data races

For purposes of determining the existence of data races, the library versions of operator \texttt{new}, user replacement versions of global operator \texttt{new}, the C standard library functions \texttt{aligned_alloc}, \texttt{calloc}, and \texttt{malloc}, the library versions of operator \texttt{delete}, user replacement versions of operator \texttt{delete}, the C standard library function \texttt{free}, and the C standard library function \texttt{realloc} shall not introduce a data race (16.4.6.10). Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before (6.9.2) the next allocation (if any) in this order.

17.6.4 Storage allocation errors

17.6.4.1 Class \texttt{bad_alloc}

```cpp
namespace std {
    class bad_alloc : public exception {
        public:
            // see 17.9.3 for the specification of the special member functions
            const char* what() const noexcept override;
    };
}
```

1 The class \texttt{bad_alloc} defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

```cpp
const char* what() const noexcept override;
```

Returns: An implementation-defined \texttt{ntbs}.

17.6.4.2 Class \texttt{bad_array_new_length}

```cpp
namespace std {
    class bad_array_new_length : public bad_alloc {
        public:
            // see 17.9.3 for the specification of the special member functions
            const char* what() const noexcept override;
    };
}
```

1 The class \texttt{bad_array_new_length} defines the type of objects thrown as exceptions by the implementation to report an attempt to allocate an array of size less than zero or greater than an implementation-defined limit (7.6.2.8).

```cpp
const char* what() const noexcept override;
```

Returns: An implementation-defined \texttt{ntbs}.

17.6.4.3 Type \texttt{new_handler}

```cpp
using new_handler = void (*)();
```

1 The type of a \textit{handler function} to be called by operator \texttt{new()} or operator \texttt{new[]()} (17.6.3) when they cannot satisfy a request for additional storage.

```cpp
Required behavior: A new_handler shall perform one of the following:
```

1 Number 1

- make more storage available for allocation and then return;
- throw an exception of type \texttt{bad_alloc} or a class derived from \texttt{bad_alloc};
- terminate execution of the program without returning to the caller.

17.6.4.4 set_new_handler

```cpp
new_handler set_new_handler(new_handler new_p) noexcept;
```

1 Effects: Establishes the function designated by new_p as the current new_handler.

```cpp
Returns: The previous new_handler.
```

1 Remarks: The initial new_handler is a null pointer.
17.6.4.5  get_new_handler

new_handler get_new_handler() noexcept;

Returns: The current new_handler.

[Note 1: This can be a null pointer value. — end note]

17.6.5  Pointer optimization barrier

template<class T> [[nodiscard]] constexpr T* launder(T* p) noexcept;

Mandates: !is_function_v<T> && !is_void_v<T> is true.

Preconditions: p represents the address A of a byte in memory. An object X that is within its lifetime (6.7.3) and whose type is similar (7.3.6) to T is located at the address A. All bytes of storage that would be reachable through the result are reachable through p (see below).

Returns: A value of type T* that points to X.

Remarks: An invocation of this function may be used in a core constant expression whenever the value of its argument may be used in a core constant expression. A byte of storage b is reachable through a pointer value that points to an object Y if there is an object Z, pointer-interconvertible with Y, such that b is within the storage occupied by Z, or the immediately-enclosing array object if Z is an array element.

[Note 1: If a new object is created in storage occupied by an existing object of the same type, a pointer to the original object can be used to refer to the new object unless its complete object is a const object or it is a base class subobject; in the latter cases, this function can be used to obtain a usable pointer to the new object. See 6.7.3. — end note]

[Example 1:
  struct X { int n; };  
  const X *p = new const X{3};  
  const int a = p->n;  
  new (const_cast<X*>(p)) const X{5};  
  // p does not point to new object (6.7.3) because its type is const  
  const int b = p->n;  
  // undefined behavior  
  const int c = std::launder(p)->n;  
  // OK  
  — end example]

17.6.6  Hardware interference size

inline constexpr size_t hardware_destructive_interference_size = implementation-defined;

This number is the minimum recommended offset between two concurrently-accessed objects to avoid additional performance degradation due to contention introduced by the implementation. It shall be at least alignof(max_align_t).

[Example 1:
  struct keep_apart {
    alignas(hardware_destructive_interference_size) atomic<int> cat;  
    alignas(hardware_destructive_interference_size) atomic<int> dog;
  };
  — end example]

inline constexpr size_t hardware_constructive_interference_size = implementation-defined;

This number is the maximum recommended size of contiguous memory occupied by two objects accessed with temporal locality by concurrent threads. It shall be at least alignof(max_align_t).

[Example 2:
  struct together {
    atomic<int> dog;  
    int puppy;
  };  
  struct kennel {
    // Other data members...  
    alignas(sizeof(together)) together pack;
  };
  — end example]
// Other data members...
};
static_assert(sizeof(together) <= hardware_constructive_interference_size);

—end example]

17.7 Type identification
[support.rtti]
17.7.1 General
[support.rtti.general]
The header `<typeinfo>` defines a type associated with type information generated by the implementation. It also defines two types for reporting dynamic type identification errors.

17.7.2 Header `<typeinfo>` synopsis
[typeinfo.syn]

```cpp
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
}
```

17.7.3 Class `type_info`
[type.info]

```cpp
namespace std {
    class type_info {
        public:
        virtual ~type_info();
        bool operator==(const type_info& rhs) const noexcept;
        bool before(const type_info& rhs) const noexcept;
        size_t hash_code() const noexcept;
        const char* name() const noexcept;
        type_info(const type_info&) = delete;  // cannot be copied
        type_info& operator=(const type_info&) = delete;  // cannot be copied
    }
}
```

1 The class `type_info` describes type information generated by the implementation (7.6.1.8). Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

```cpp
bool operator==(const type_info& rhs) const noexcept;
```

2 Effects: Compares the current object with `rhs`.

3 `Returns`: `true` if the two values describe the same type.

```cpp
bool before(const type_info& rhs) const noexcept;
```

4 Effects: Compares the current object with `rhs`.

5 `Returns`: `true` if `*this` precedes `rhs` in the implementation’s collation order.

```cpp
size_t hash_code() const noexcept;
```

6 `Returns`: An unspecified value, except that within a single execution of the program, it shall return the same value for any two `type_info` objects which compare equal.

7 Remarks: An implementation should return different values for two `type_info` objects which do not compare equal.

```cpp
const char* name() const noexcept;
```

8 `Returns`: An implementation-defined `ntbs`.

9 Remarks: The message may be a null-terminated multibyte string (16.3.3.5.3), suitable for conversion and display as a `wstring` (21.3, 28.4.2.5).
17.7.4 Class bad_cast

namespace std {
    class bad_cast : public exception {
        public:
            // see 17.9.3 for the specification of the special member functions
            const char* what() const noexcept override;
    };
}

The class bad_cast defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid dynamic_cast expression (7.6.1.7).

const char* what() const noexcept override;

Returns: An implementation-defined NTBS.

17.7.5 Class bad_typeid

namespace std {
    class bad_typeid : public exception {
        public:
            // see 17.9.3 for the specification of the special member functions
            const char* what() const noexcept override;
    };
}

The class bad_typeid defines the type of objects thrown as exceptions by the implementation to report a null pointer in a typeid expression (7.6.1.8).

const char* what() const noexcept override;

Returns: An implementation-defined NTBS.

17.8 Source location

17.8.1 Header <source_location> synopsis

The header <source_location> defines the class source_location that provides a means to obtain source location information.

namespace std {
    struct source_location;
}

17.8.2 Class source_location

17.8.2.1 General

namespace std {
    struct source_location {
        // source location construction
        static constexpr source_location current() noexcept;
        constexpr source_location() noexcept;

        // source location field access
        constexpr uint_least32_t line() const noexcept;
        constexpr uint_least32_t column() const noexcept;
        constexpr const char* file_name() const noexcept;
        constexpr const char* function_name() const noexcept;

        private:
            uint_least32_t line_;  // exposition only
            uint_least32_t column_; // exposition only
            const char* file_name_; // exposition only
            const char* function_name_; // exposition only
    };
}
The type `source_location` meets the `Cpp17DefaultConstructible`, `Cpp17CopyConstructible`, `Cpp17CopyAssignable`, and `Cpp17Destructible` requirements (16.4.4.2). All of the following conditions are true:

(1.1) \[ \text{is_nothrow_move_constructible_v<source_location>} \]
(1.2) \[ \text{is_nothrow_move_assignable_v<source_location>} \]
(1.3) \[ \text{is_nothrow_swappable_v<source_location>} \]

[Note 1: The intent of `source_location` is to have a small size and efficient copying. It is unspecified whether the copy/move constructors and the copy/move assignment operators are trivial and/or constexpr. — end note]

The data members `file_name_` and `function_name_` always each refer to an NTBS.

The copy/move constructors and the copy/move assignment operators of `source_location` meet the following postconditions: Given two objects `lhs` and `rhs` of type `source_location`, where `lhs` is a copy/move result of `rhs`, and where `rhs_p` is a value denoting the state of `rhs` before the corresponding copy/move operation, then each of the following conditions is true:

(3.1) \[ \text{strcmp(lhs.file_name(), rhs_p.file_name()) == 0} \]
(3.2) \[ \text{strcmp(lhs.function_name(), rhs_p.function_name()) == 0} \]
(3.3) \[ \text{lhs.line() == rhs_p.line()} \]
(3.4) \[ \text{lhs.column() == rhs_p.column()} \]

17.8.2.2 Creation

```cpp
static constexpr source_location current() noexcept;
```

Returns:

(1.1) When invoked by a function call whose `postfix-expression` is a (possibly parenthesized) `id-expression` naming `current`, returns a `source_location` with an implementation-defined value. The value should be affected by `#line` (15.7) in the same manner as for `__LINE__` and `__FILE__`. The values of the exposition-only data members of the returned `source_location` object are indicated in Table 38.

Table 38: Value of object returned by `current` [tab:support.srcloc.current]

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>line_</code></td>
<td>A presumed line number (15.11). Line numbers are presumed to be 1-indexed; however, an implementation is encouraged to use 0 when the line number is unknown.</td>
</tr>
<tr>
<td><code>column_</code></td>
<td>An implementation-defined value denoting some offset from the start of the line denoted by <code>line_</code>. Column numbers are presumed to be 1-indexed; however, an implementation is encouraged to use 0 when the column number is unknown.</td>
</tr>
<tr>
<td><code>file_name_</code></td>
<td>A presumed name of the current source file (15.11) as an NTBS.</td>
</tr>
<tr>
<td><code>function_name_</code></td>
<td>A name of the current function such as in <code>__func__</code> (9.5.1) if any, an empty string otherwise.</td>
</tr>
</tbody>
</table>

(1.2) Otherwise, when invoked in some other way, returns a `source_location` whose data members are initialized with valid but unspecified values.

Remarks: Any call to `current` that appears as a default member initializer (11.4), or as a subexpression thereof, should correspond to the location of the constructor definition or aggregate initialization that uses the default member initializer. Any call to `current` that appears as a default argument (9.3.4.7), or as a subexpression thereof, should correspond to the location of the invocation of the function that uses the default argument (7.6.1.3).
Example 1:

```cpp
struct s {
    source_location member = source_location::current();
    int other_member;
    s(source_location loc = source_location::current())
        : member(loc) // values of member refer to the location of the calling function (9.3.4.7)
    {} s(int blather) :
        // values of member refer to this location
        other_member(blather)
    {} s(double)
        // values of member refer to this location
    {};
}

void f(source_location a = source_location::current()) {
    source_location b = source_location::current(); // values in b refer to this line
}

void g() {
    f(); // f's first argument corresponds to this line of code
    source_location c = source_location::current();
    f(c); // f's first argument gets the same values as c, above
}
```

— end example]

cconstexpr source_location() noexcept;

Effects: The data members are initialized with valid but unspecified values.

17.8.2.3 Observers [support.srcloc.obs]
cconstexpr uint_least32_t line() const noexcept;
    Returns: line_
cconstexpr uint_least32_t column() const noexcept;
    Returns: column_
cconstexpr const char* file_name() const noexcept;
    Returns: file_name_
cconstexpr const char* function_name() const noexcept;
    Returns: function_name_

17.9 Exception handling [support.exception]

17.9.1 General [support.exception.general]

The header `<exception>` defines several types and functions related to the handling of exceptions in a C++ program.

17.9.2 Header `<exception>` synopsis [exception.syn]

```cpp
namespace std {
    class exception;
    class bad_exception;
    class nested_exception;

    using terminate_handler = void (*)();
    terminate_handler get_terminate() noexcept;
    terminate_handler set_terminate(terminate_handler f) noexcept;
    [[noreturn]] void terminate() noexcept;
    int uncaught_exceptions() noexcept;

    using exception_ptr = unspecified;
```
exception_ptr current_exception() noexcept;
[[noreturn]] void rethrow_exception(exception_ptr p);
template<class E> exception_ptr make_exception_ptr(E e) noexcept;

[[noreturn]] void throw_with_nested(T&& t);
template<class E> void rethrow_if_nested(const E& e);

17.9.3 Class exception

namespace std {
    class exception {
    public:
        exception() noexcept;
        exception(const exception&) noexcept;
        exception& operator=(const exception&) noexcept;
        virtual ~exception();
        virtual const char* what() const noexcept;
    };
}

1 The class exception defines the base class for the types of objects thrown as exceptions by C++ standard library components, and certain expressions, to report errors detected during program execution.

2 Each standard library class T that derives from class exception has the following publicly accessible member functions, each of them having a non-throwing exception specification (14.5):
   (2.1) — default constructor (unless the class synopsis shows other constructors)
   (2.2) — copy constructor
   (2.3) — copy assignment operator

The copy constructor and the copy assignment operator meet the following postcondition: If two objects \( \text{lhs} \) and \( \text{rhs} \) both have dynamic type \( T \) and \( \text{lhs} \) is a copy of \( \text{rhs} \), then \( \text{strcmp} \left( \text{lhs}.\text{what}(), \text{rhs}.\text{what}() \right) \) is equal to 0. The what() member function of each such \( T \) satisfies the constraints specified for exception::what() (see below).

exception(const exception& rhs) noexcept;
exception& operator=(const exception& rhs) noexcept;

3 Postconditions: If *this and \( \text{rhs} \) both have dynamic type exception then the value of the expression \( \text{strcmp} \left( \text{what}(), \text{rhs}.\text{what}() \right) \) shall equal 0.

virtual ~exception();

4 Effects: Destroys an object of class exception.

virtual const char* what() const noexcept;

5 Returns: An implementation-defined NTBS.

6 Remarks: The message may be a null-terminated multibyte string (16.3.3.5.3), suitable for conversion and display as a wstring (21.3, 28.4.2.5). The return value remains valid until the exception object from which it is obtained is destroyed or a non-const member function of the exception object is called.

17.9.4 Class bad_exception

namespace std {
    class bad_exception : public exception {
    public:
        // see 17.9.3 for the specification of the special member functions
        const char* what() const noexcept override;
    };
}

1 The class bad_exception defines the type of the object referenced by the exception_ptr returned from a call to current_exception (17.9.7) when the currently active exception object fails to copy.

const char* what() const noexcept override;

2 Returns: An implementation-defined NTBS.
17.9.5 Abnormal termination

### 17.9.5.1 Type terminate_handler

using terminate_handler = void (*)(void);

1. The type of a *handler function* to be called by `std::terminate()` when terminating exception processing.

2. Required behavior: A `terminate_handler` shall terminate execution of the program without returning to the caller.

3. Default behavior: The implementation's default `terminate_handler` calls `abort()`.

### 17.9.5.2 set_terminate

`terminate_handler set_terminate(terminate_handler f)` noexcept;

1. Effects: Establishes the function designated by `f` as the current handler function for terminating exception processing.

2. Returns: The previous `terminate_handler`.

3. Remarks: It is unspecified whether a null pointer value designates the default `terminate_handler`.

### 17.9.5.3 get_terminate

`terminate_handler get_terminate()` noexcept;

1. Returns: The current `terminate_handler`.

   [Note 1: This can be a null pointer value. — end note]

### 17.9.5.4 terminate

`[noreturn] void terminate()` noexcept;

1. Effects: Calls a `terminate_handler` function. It is unspecified which `terminate_handler` function will be called if an exception is active during a call to `set_terminate`. Otherwise calls the current `terminate_handler` function.

   [Note 1: A default `terminate_handler` is always considered a callable handler in this context. — end note]

2. Remarks: Called by the implementation when exception handling must be abandoned for any of several reasons (14.6.2). May also be called directly by the program.

### 17.9.6 uncaught_exceptions

`int uncaught_exceptions()` noexcept;

1. Returns: The number of uncaught exceptions (14.6.3).

2. Remarks: When `uncaught_exceptions()` > 0, throwing an exception can result in a call of the function `std::terminate` (14.6.2).

### 17.9.7 Exception propagation

using exception_ptr = unspecified;

1. The type `exception_ptr` can be used to refer to an exception object.

2. `exception_ptr` meets the requirements of `Cpp17NullablePointer` (Table 33).

3. Two non-null values of type `exception_ptr` are equivalent and compare equal if and only if they refer to the same exception.

4. The default constructor of `exception_ptr` produces the null value of the type.

5. `exception_ptr` shall not be implicitly convertible to any arithmetic, enumeration, or pointer type.

   [Note 1: An implementation might use a reference-counted smart pointer as `exception_ptr`. — end note]

6. For purposes of determining the presence of a data race, operations on `exception_ptr` objects shall access and modify only the `exception_ptr` objects themselves and not the exceptions they refer to. Use of `rethrow_exception` on `exception_ptr` objects that refer to the same exception object shall not introduce a data race.
exception_ptr current_exception() noexcept;

Returns: An exception_ptr object that refers to the currently handled exception (14.4) or a copy of the currently handled exception, or a null exception_ptr object if no exception is being handled. The referenced object shall remain valid at least as long as there is an exception_ptr object that refers to it. If the function needs to allocate memory and the attempt fails, it returns an exception_ptr object that refers to an instance of bad_alloc. It is unspecified whether the return values of two successive calls to current_exception refer to the same exception object.

Note 3: That is, it is unspecified whether current_exception creates a new copy each time it is called. — end note

If the attempt to copy the current exception object throws an exception, the function returns an exception_ptr object that refers to the thrown exception or, if this is not possible, to an instance of bad_exception.

Note 4: The copy constructor of the thrown exception can also fail, so the implementation is allowed to substitute a bad_exception object to avoid infinite recursion. — end note

[[noreturn]] void rethrow_exception(exception_ptr p);

Preconditions: p is not a null pointer.

template<class E> exception_ptr make_exception_ptr(E e) noexcept;

Effects: Creates an exception_ptr object that refers to a copy of e, as if:

```cpp
try {
  throw e;
} catch(...) {
  return current_exception();
}
```

Note 5: This function is provided for convenience and efficiency reasons. — end note

17.9.8 nested_exception

namespace std {

class nested_exception {
  public:
    nested_exception() noexcept;
    nested_exception(const nested_exception&) noexcept = default;
    nested_exception& operator=(const nested_exception&) noexcept = default;
    virtual ~nested_exception() = default;

    // access functions
    [[noreturn]] void rethrow_nested() const;
    exception_ptr nested_ptr() const noexcept;
  };

  template<class T> [[noreturn]] void throw_with_nested(T&& t);
  template<class E> void rethrow_if_nested(const E& e);
};

The class nested_exception is designed for use as a mixin through multiple inheritance. It captures the currently handled exception and stores it for later use.

Note 1: nested_exception has a virtual destructor to make it a polymorphic class. Its presence can be tested for with dynamic_cast. — end note

nested_exception() noexcept;

Effects: The constructor calls current_exception() and stores the returned value.
[[noreturn]] void rethrow_nested() const;

Effects: If nested_ptr() returns a null pointer, the function calls the function std::terminate. Otherwise, it throws the stored exception captured by *this.

exception_ptr nested_ptr() const noexcept;

Returns: The stored exception captured by this nested_exception object.

template<class T> [[noreturn]] void throw_with_nested(T&& t);

Let U be decay_t<T>.

Preconditions: U meets the Cpp17CopyConstructible requirements.

Throws: If is_class_v<U> && !is_final_v<U> && !is_base_of_v<nested_exception, U> is true, an exception of unspecified type that is publicly derived from both U and nested_exception and constructed from std::forward<T>(t), otherwise std::forward<T>(t).

template<class E> void rethrow_if_nested(const E& e);

Effects: If E is not a polymorphic class type, or if nested_exception is an inaccessible or ambiguous base class of E, there is no effect. Otherwise, performs:

if (auto p = dynamic_cast<const nested_exception*>(addressof(e)))
    p->rethrow_nested();

17.10 Initializer lists

17.10.1 General

The header <initializer_list> defines a class template and several support functions related to list-initialization (see 9.4.5). All functions specified in 17.10 are signal-safe (17.13.5).

17.10.2 Header <initializer_list> synopsis

namespace std {
    template<class E> class initializer_list {
        public:
            using value_type = E;
            using reference = const E&;
            using const_reference = const E&;
            using size_type = size_t;
            using iterator = const E*;
            using const_iterator = const E*;

            constexpr initializer_list() noexcept;
            constexpr size_t size() const noexcept; // number of elements
            constexpr const E* begin() const noexcept; // first element
            constexpr const E* end() const noexcept; // one past the last element
        };

        // 17.10.5, initializer list range access
        template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;
        template<class E> constexpr const E* end(initializer_list<E> il) noexcept;
    };

} // 17.10.5, initializer list range access

An object of type initializer_list<E> provides access to an array of objects of type const E.

[Note 1: A pair of pointers or a pointer plus a length would be obvious representations for initializer_list. initializer_list is used to implement initializer lists as specified in 9.4.5. Copying an initializer list does not copy the underlying elements. — end note]

If an explicit specialization or partial specialization of initializer_list is declared, the program is ill-formed.
17.10.3 Initializer list constructors [support.initlist.cons]

constexpr initializer_list() noexcept;

Postconditions: size() == 0.

17.10.4 Initializer list access [support.initlist.access]

constexpr const E* begin() const noexcept;

Returns: A pointer to the beginning of the array. If size() == 0 the values of begin() and end() are unspecified but they shall be identical.

constexpr const E* end() const noexcept;

Returns: begin() + size().

constexpr size_t size() const noexcept;

Returns: The number of elements in the array.

Complexity: Constant time.

17.10.5 Initializer list range access [support.initlist.range]

template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;

Returns: il.begin().

template<class E> constexpr const E* end(initializer_list<E> il) noexcept;

Returns: il.end().

17.11 Comparisons [cmp]

17.11.1 Header <compare> synopsis [compare.syn]

The header <compare> specifies types, objects, and functions for use primarily in connection with the three-way comparison operator (7.6.8).

namespace std {
    // 17.11.2, comparison category types
    class partial_ordering;
    class weak_ordering;
    class strong_ordering;

    // named comparison functions
    constexpr bool is_eq (partial_ordering cmp) noexcept { return cmp == 0; }
    constexpr bool is_neq (partial_ordering cmp) noexcept { return cmp != 0; }
    constexpr bool is_lt (partial_ordering cmp) noexcept { return cmp < 0; }
    constexpr bool is_lteq (partial_ordering cmp) noexcept { return cmp <= 0; }
    constexpr bool is_gt (partial_ordering cmp) noexcept { return cmp > 0; }
    constexpr bool is_gteq (partial_ordering cmp) noexcept { return cmp >= 0; }

    // 17.11.3, common comparison category type
    template<class... Ts>
    struct common_comparison_category {
        using type = see below;
    };
    template<class... Ts>
    using common_comparison_category_t = typename common_comparison_category<Ts...>::type;

    // 17.11.4, concept three_way_comparable
    template<class T, class Cat = partial_ordering>
    concept three_way_comparable = see below;
    template<class T, class U, class Cat = partial_ordering>
    concept three_way_comparable_with = see below;

    // 17.11.5, result of three-way comparison
    template<class T, class U = T> struct compare_three_way_result;
template<
    class T, class U = T>
using compare_three_way_result_t = typename compare_three_way_result<T, U>::type;

// 20.14.8.8, class compare_three_way
struct compare_three_way;

// 17.11.6, comparison algorithms
inline namespace unspecified {
    inline constexpr unspecified strong_order = unspecified;
    inline constexpr unspecified weak_order = unspecified;
    inline constexpr unspecified partial_order = unspecified;
    inline constexpr unspecified compare_strong_order_fallback = unspecified;
    inline constexpr unspecified compare_weak_order_fallback = unspecified;
    inline constexpr unspecified compare_partial_order_fallback = unspecified;
}

17.11.2 Comparison category types

17.11.2.1 Preamble

The types partial_ordering, weak_ordering, and strong_ordering are collectively termed the comparison category types. Each is specified in terms of an exposition-only data member named value whose value typically corresponds to that of an enumerator from one of the following exposition-only enumerations:

```cpp
enum class ord {
    equal = 0, equivalent = equal, less = -1, greater = 1 }; // exposition only
enum class ncmp { unordered = -127 }; // exposition only
```

[Note 1: The type strong_ordering corresponds to the term total ordering in mathematics. — end note]

The relational and equality operators for the comparison category types are specified with an anonymous parameter of unspecified type. This type shall be selected by the implementation such that these parameters can accept literal 0 as a corresponding argument.

[Example 1: nullptr_t meets this requirement. — end example]

In this context, the behavior of a program that supplies an argument other than a literal 0 is undefined.

For the purposes of subclause 17.11.2, substitutability is the property that \( f(a) == f(b) \) is true whenever \( a == b \) is true, where \( f \) denotes a function that reads only comparison-salient state that is accessible via the argument’s public const members.

17.11.2.2 Class partial_ordering

The partial_ordering type is typically used as the result type of a three-way comparison operator (7.6.8) that (a) admits all of the six two-way comparison operators (7.6.9, 7.6.10), (b) does not imply substitutability, and (c) permits two values to be incomparable.220

```cpp
namespace std {
    class partial_ordering {
        int value; // exposition only
        bool is_ordered; // exposition only

        // exposition-only constructors
        constexpr explicit partial_ordering(ord v) noexcept : value(int(v)), is_ordered(true) {} // exposition only
        constexpr explicit partial_ordering(ncmp v) noexcept : value(int(v)), is_ordered(false) {} // exposition only

        public:
            // valid values
            static const partial_ordering less;
            static const partial_ordering equivalent;
            static const partial_ordering greater;
            static const partial_ordering unordered;

    }
}
```

220) That is, \( a < b \), \( a == b \), and \( a > b \) might all be false.
17.11.2.3 Class weak_ordering

The weak_ordering type is typically used as the result type of a three-way comparison operator (7.6.8) that (a) admits all of the six two-way comparison operators (7.6.9, 7.6.10), and (b) does not imply substitutability.

namespace std {
  class weak_ordering {
    int value; // exposition only

    // exposition-only constructors
    constexpr explicit weak_ordering(ord v) noexcept : value(int(v)) {} // exposition only

public:
  // valid values
  static const weak_ordering less;
  static const weak_ordering equivalent;
  static const weak_ordering greater;

  // conversions
  constexpr operator partial_ordering() const noexcept;

  // comparisons
  friend constexpr bool operator==(weak_ordering v, unspecified) noexcept;

  friend constexpr bool operator==(weak_ordering v, partial_ordering w) noexcept = default;
  friend constexpr bool operator<(weak_ordering v, unspecified) noexcept;
  friend constexpr bool operator<(weak_ordering v, partial_ordering w) noexcept;
  friend constexpr bool operator>(weak_ordering v, unspecified) noexcept;
  friend constexpr bool operator>(weak_ordering v, partial_ordering w) noexcept;
  friend constexpr bool operator<=(weak_ordering v, unspecified) noexcept;
  friend constexpr bool operator<=(weak_ordering v, partial_ordering w) noexcept;
  friend constexpr bool operator>=(weak_ordering v, unspecified) noexcept;
  friend constexpr bool operator>=(weak_ordering v, partial_ordering w) noexcept;

  constexpr partial_ordering operator<=>(weak_ordering v, unspecified) noexcept;
  constexpr partial_ordering operator<=>(unspecified, weak_ordering v) noexcept;

2 Returns: For operator@, v.is_ordered && v.value @ 0.

  constexpr bool operator<(unspecified, partial_ordering v) noexcept;
  constexpr bool operator<(unspecified, partial_ordering w) noexcept = default;
  friend constexpr bool operator<(unspecified, unspecified) noexcept;
  friend constexpr bool operator<(unspecified, partial_ordering v) noexcept;
  friend constexpr bool operator>(unspecified, partial_ordering v) noexcept;
  friend constexpr bool operator>(unspecified, partial_ordering w) noexcept;
  friend constexpr bool operator<=(unspecified, unspecified) noexcept;
  friend constexpr bool operator<=(unspecified, partial_ordering v) noexcept;
  friend constexpr bool operator>=(unspecified, unspecified) noexcept;
  friend constexpr bool operator>=(unspecified, partial_ordering v) noexcept;

3 Returns: For operator@, v.is_ordered && 0 @ v.value.

  constexpr bool operator<=(unspecified, partial_ordering v) noexcept;
  constexpr bool operator<=(unspecified, partial_ordering w) noexcept = default;
  friend constexpr bool operator<=(unspecified, unspecified) noexcept;
  friend constexpr bool operator<=(unspecified, partial_ordering v) noexcept;
  friend constexpr bool operator>=(unspecified, unspecified) noexcept;
  friend constexpr bool operator>=(unspecified, partial_ordering v) noexcept;

4 Returns: v.

  constexpr bool operator>=(unspecified, partial_ordering v) noexcept;
  friend constexpr partial_ordering operator<=>(unspecified, partial_ordering v) noexcept;

5 Returns: v < 0 ? partial_ordering::greater : v > 0 ? partial_ordering::less : v.
friend constexpr bool operator==(weak_ordering v, weak_ordering w) noexcept = default;
friend constexpr bool operator<(weak_ordering v, unspecified) noexcept;
friend constexpr bool operator<=(weak_ordering v, unspecified) noexcept;
friend constexpr bool operator>(weak_ordering v, unspecified) noexcept;
friend constexpr bool operator>=(weak_ordering v, unspecified) noexcept;
friend constexpr bool operator< (unspecified, weak_ordering v) noexcept;
friend constexpr bool operator> (unspecified, weak_ordering v) noexcept;
friend constexpr bool operator<= (unspecified, weak_ordering v) noexcept;
friend constexpr bool operator>= (unspecified, weak_ordering v) noexcept;
friend constexpr weak_ordering operator<=>(weak_ordering v, unspecified) noexcept;
friend constexpr weak_ordering operator<=>(unspecified, weak_ordering v) noexcept;

constexpr operator partial_ordering() const noexcept;

Returns:
- value == 0 ? partial_ordering::equivalent :
- value < 0 ? partial_ordering::less :
- partial_ordering::greater

constexpr bool operator==(weak_ordering v, unspecified) noexcept;
constexpr bool operator<(weak_ordering v, unspecified) noexcept;
constexpr bool operator>(weak_ordering v, unspecified) noexcept;
constexpr bool operator<=(weak_ordering v, unspecified) noexcept;
constexpr bool operator>=(weak_ordering v, unspecified) noexcept;

Returns: v.value @ 0 for operator@.

constexpr bool operator<(unspecified, weak_ordering v) noexcept;
constexpr bool operator>(unspecified, weak_ordering v) noexcept;
constexpr bool operator<=(unspecified, weak_ordering v) noexcept;
constexpr bool operator>=(unspecified, weak_ordering v) noexcept;

Returns: 0 @ v.value for operator@.

constexpr weak_ordering operator<=(weak_ordering v, unspecified) noexcept;

Returns: v.

constexpr weak_ordering operator>=(unspecified, weak_ordering v) noexcept;

Returns: v < 0 ? weak_ordering::greater : v > 0 ? weak_ordering::less : v.

17.11.2.4 Class strong_ordering

The strong_ordering type is typically used as the result type of a three-way comparison operator (7.6.8) that (a) admits all of the six two-way comparison operators (7.6.9, 7.6.10), and (b) does imply substitutability.
// conversions
constexpr operator partial_ordering() const noexcept;
constexpr operator weak_ordering() const noexcept;

// comparisons
friend constexpr bool operator==(strong_ordering v, unspecified) noexcept;
friend constexpr bool operator==(strong_ordering v, strong_ordering w) noexcept = default;
friend constexpr bool operator<(strong_ordering v, unspecified) noexcept;
friend constexpr bool operator<(strong_ordering v, strong_ordering w) noexcept = default;
friend constexpr bool operator<=(strong_ordering v, unspecified) noexcept;
friend constexpr bool operator<=(strong_ordering v, strong_ordering w) noexcept = default;
friend constexpr bool operator>(unspecified, strong_ordering v) noexcept;
friend constexpr bool operator>(unspecified, strong_ordering v) noexcept = default;
friend constexpr bool operator>=(unspecified, strong_ordering v) noexcept;
friend constexpr bool operator>=(unspecified, strong_ordering v) noexcept = default;
friend constexpr strong_ordering operator<=>(strong_ordering v, unspecified) noexcept;
friend constexpr strong_ordering operator<=>(unspecified, strong_ordering v) noexcept;

// valid values' definitions
inline constexpr strong_ordering strong_ordering::less(ord::less);
inline constexpr strong_ordering strong_ordering::equal(ord::equal);
inline constexpr strong_ordering strong_ordering::equivalent(ord::equivalent);
inline constexpr strong_ordering strong_ordering::greater(ord::greater);

17.11.3 Class template common_comparison_category
The type common_comparison_category provides an alias for the strongest comparison category to which all of the template arguments can be converted.
template<class... Ts>
struct common_comparison_category {
    using type = see below;
};

Remarks: The member typedef-name type denotes the common comparison type (11.11.3) of Ts..., the expanded parameter pack, or void if any element of Ts is not a comparison category type.

[Note 2: This is std::strong_ordering if the expansion is empty. — end note]

17.11.4 Concept three_way_comparable

Let t and u be lvalues of types const remove_reference_t<T> and const remove_reference_t<U>, respectively. T and U model partially-ordered-with<T, U> only if:

1. t < u, t <= u, t == u, t > u, t == t, u > t, and u >= t have the same domain.
2. bool(t < u) == bool(u > t) is true,
3. bool(u < t) == bool(t > u) is true,
4. bool(t <= u) == bool(u >= t) is true, and
5. bool(u <= t) == bool(t >= u) is true.

Let a and b be lvalues of type const remove_reference_t<T>. T and Cat model three_way_comparable<T, Cat> only if:

1. (a == b) == bool(a == b) is true,
2. (a != b) == bool(a != b) is true,
3. (a == b) == bool(a == b) and (0 == (b <= a)) are equal,
4. (a == b == 0) == bool(a <= b) is true,
5. (a <= b) == bool(a > b) is true,
6. (a <= b) == bool(a <= b) is true,
7. (a <= b == 0) == bool(a >= b) is true, and
8. if Cat is convertible to strong_ordering, T models totally_ordered (18.5.4).
template<class T, class U, class Cat = partial_ordering>
concept three_way_comparable_with =
    three_way_comparable<T, Cat> &&
    three_way_comparable<U, Cat> &&
    common_reference_with<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
    three_way_comparable<
        common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>, Cat> &&
    weakly-equality-comparable-with<T, U> &&
    partially-ordered-with<T, U> &&
requires(const remove_reference_t<T>& t, const remove_reference_t<U>& u) { { t <=> u } -> compares-as<Cat>; { u <=> t } -> compares-as<Cat>; };

Let t and u be lvalues of types const remove_reference_t<T> and const remove_reference_t<U>, respectively. Let C be common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>. T, U, and Cat model three_way_comparable_with<T, U, Cat> only if:

(3.1) — t <= u and u <= t have the same domain,
(3.2) — ((t <= u) <=> 0) and (0 <=> (u <= t)) are equal,
(3.3) — (t <= u == 0) == bool(t == u) is true,
(3.4) — (t <= u != 0) == bool(t != u) is true,
(3.5) — Cat(t <= u) == Cat(C(t) <=> C(u)) is true,
(3.6) — (t <= u < 0) == bool(t < u) is true,
(3.7) — (t <= u > 0) == bool(t > u) is true,
(3.8) — (t <= u <= 0) == bool(t <= u) is true,
(3.9) — (t <= u >= 0) == bool(t >= u) is true, and
(3.10) — if Cat is convertible to strong_ordering, T and U model totally_ordered_with<T, U> (18.5.4).

17.11.5 Result of three-way comparison [cmp.result]

The behavior of a program that adds specializations for the compare_three_way_result template defined in this subclause is undefined.

For the compare_three_way_result type trait applied to the types T and U, let t and u denote lvalues of types const remove_reference_t<T> and const remove_reference_t<U>, respectively. If the expression t <=> u is well-formed when treated as an unevaluated operand (7.2.3), the member typedef-name type denotes the type decltype(t <=> u). Otherwise, there is no member type.

17.11.6 Comparison algorithms [cmp.alg]

The name strong_order denotes a customization point object (16.3.3.3.6). Given subexpressions E and F, the expression strong_order(E, F) is expression-equivalent (3.21) to the following:

(1.1) — If the decayed types of E and F differ, strong_order(E, F) is ill-formed.
(1.2) — Otherwise, strong_ordering(strong_order(E, F)) if it is a well-formed expression with overload resolution performed in a context that does not include a declaration of std::strong_order.
(1.3) — Otherwise, if the decayed type T of E is a floating-point type, yields a value of type strong_ordering that is consistent with the ordering observed by T’s comparison operators, and if numeric_limits<T>::is_iec559 is true, is additionally consistent with the totalOrder operation as specified in ISO/IEC/IEEE 60559.
(1.4) — Otherwise, strong_ordering(compare_three_way()(E, F)) if it is a well-formed expression.
(1.5) — Otherwise, strong_order(E, F) is ill-formed.

[Note 1: This case can result in substitution failure when strong_order(E, F) appears in the immediate context of a template instantiation. — end note]

The name weak_order denotes a customization point object (16.3.3.3.6). Given subexpressions E and F, the expression weak_order(E, F) is expression-equivalent (3.21) to the following:

(2.1) — If the decayed types of E and F differ, weak_order(E, F) is ill-formed.
Otherwise, \( \text{weak\_ordering}(\text{weak\_order}(E, F)) \) if it is a well-formed expression with overload resolution performed in a context that does not include a declaration of \( \text{std}::\text{weak\_order} \).

Otherwise, if the decayed type \( T \) of \( E \) is a floating-point type, yields a value of type \( \text{weak\_ordering} \) that is consistent with the ordering observed by \( T \)'s comparison operators and \( \text{strong\_order} \), and if \( \text{numeric\_limits}<T>::\text{is\_iec559} \) is true, is additionally consistent with the following equivalence classes, ordered from lesser to greater:

- together, all negative NaN values;
- negative infinity;
- each normal negative value;
- each subnormal negative value;
- together, both zero values;
- each subnormal positive value;
- each normal positive value;
- positive infinity;
- together, all positive NaN values.

Otherwise, \( \text{weak\_ordering}(\text{compare\_three\_way}()(E, F)) \) if it is a well-formed expression.

Otherwise, \( \text{weak\_order}(E, F) \) is ill-formed.

[Note 2: This case can result in substitution failure when \( \text{std}::\text{weak\_order}(E, F) \) appears in the immediate context of a template instantiation. — end note]

3 The name \( \text{partial\_order} \) denotes a customization point object (16.3.3.3.6). Given subexpressions \( E \) and \( F \), the expression \( \text{partial\_order}(E, F) \) is expression-equivalent (3.21) to the following:

- If the decayed types of \( E \) and \( F \) differ, \( \text{partial\_order}(E, F) \) is ill-formed.
- Otherwise, \( \text{partial\_ordering}(\text{partial\_order}(E, F)) \) if it is a well-formed expression with overload resolution performed in a context that does not include a declaration of \( \text{std}::\text{partial\_order} \).
- Otherwise, \( \text{partial\_ordering}(\text{compare\_three\_way}()(E, F)) \) if it is a well-formed expression.
- Otherwise, \( \text{partial\_order}(E, F) \) is ill-formed.

[Note 3: This case can result in substitution failure when \( \text{std}::\text{partial\_order}(E, F) \) appears in the immediate context of a template instantiation. — end note]

4 The name \( \text{compare\_strong\_order\_fallback} \) denotes a customization point object (16.3.3.3.6). Given subexpressions \( E \) and \( F \), the expression \( \text{compare\_strong\_order\_fallback}(E, F) \) is expression-equivalent (3.21) to:

- If the decayed types of \( E \) and \( F \) differ, \( \text{compare\_strong\_order\_fallback}(E, F) \) is ill-formed.
- Otherwise, \( \text{strong\_order}(E, F) \) if it is a well-formed expression.
- Otherwise, if the expressions \( E == F \) and \( E < F \) are both well-formed and convertible to \( \text{bool} \),

\[
E == F \ ? \ \text{strong\_ordering}::\text{equal} : \\
E < F \ ? \ \text{strong\_ordering}::\text{less} : \\
\text{strong\_ordering}::\text{greater}
\]

except that \( E \) and \( F \) are evaluated only once.

- Otherwise, \( \text{compare\_strong\_order\_fallback}(E, F) \) is ill-formed.

5 The name \( \text{compare\_weak\_order\_fallback} \) denotes a customization point object (16.3.3.3.6). Given subexpressions \( E \) and \( F \), the expression \( \text{compare\_weak\_order\_fallback}(E, F) \) is expression-equivalent (3.21) to:

- If the decayed types of \( E \) and \( F \) differ, \( \text{compare\_weak\_order\_fallback}(E, F) \) is ill-formed.
- Otherwise, \( \text{weak\_order}(E, F) \) if it is a well-formed expression.
- Otherwise, if the expressions \( E == F \) and \( E < F \) are both well-formed and convertible to \( \text{bool} \),
E == F ? weak_ordering::equivalent :
E < F ? weak_ordering::less :
weak_ordering::greater
except that E and F are evaluated only once.

— Otherwise, compare_weak_order_fallback(E, F) is ill-formed.

6 The name compare_partial_order_fallback denotes a customization point object (16.3.3.6). Given subexpressions E and F, the expression compare_partial_order_fallback(E, F) is expression-equivalent (3.21) to:

— If the decayed types of E and F differ, compare_partial_order_fallback(E, F) is ill-formed.
— Otherwise, partial_order(E, F) if it is a well-formed expression.
— Otherwise, if the expressions E == F and E < F are both well-formed and convertible to bool,

  E == F ? partial_ordering::equivalent :
  E < F ? partial_ordering::less :
  F < E ? partial_ordering::greater :
    partial_ordering::unordered
except that E and F are evaluated only once.

— Otherwise, compare_partial_order_fallback(E, F) is ill-formed.

17.12 Coroutines

17.12.1 General

The header <coroutine> defines several types providing compile and run-time support for coroutines in a C++ program.

17.12.2 Header <coroutine> synopsis

```
#include <compare>  // see 17.11.1

namespace std {
  // 17.12.3, coroutine traits
  template<class R, class... ArgTypes>
  struct coroutine_traits;

  // 17.12.4, coroutine handle
  template<class Promise = void>
  struct coroutine_handle;

  // 17.12.4.7, comparison operators
  constexpr bool operator==(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr strong_ordering operator<=>(coroutine_handle<> x, coroutine_handle<> y) noexcept;

  // 17.12.4.8, hash support
  template<class T> struct hash;
  template<class P> struct hash<coroutine_handle<P>>;

  // 17.12.5, no-op coroutines
  struct noop_coroutine_promise;
  template<> struct coroutine_handle<noop_coroutine_promise>;
  using noop_coroutine_handle = coroutine_handle<noop_coroutine_promise>;
  noop_coroutine_handle noop_coroutine() noexcept;

  // 17.12.6, trivial awaitables
  struct suspend_never;
  struct suspend_always;
}
```
17.12.3 Coroutine traits

17.12.3.1 General

Subclause 17.12.3 defines requirements on classes representing coroutine traits, and defines the class template coroutine_traits that meets those requirements.

17.12.3.2 Class template coroutine_traits

The header <coroutine> defines the primary template coroutine_traits such that if ArgTypes is a parameter pack of types and if the qualified-id R::promise_type is valid and denotes a type (13.10.3), then coroutine_traits<R,ArgTypes...> has the following publicly accessible member:

```cpp
using promise_type = typename R::promise_type;
```

Otherwise, coroutine_traits<R,ArgTypes...> has no members.

Program-defined specializations of this template shall define a publicly accessible nested type named promise_type.

17.12.4 Class template coroutine_handle

17.12.4.1 General

namespace std {

struct coroutine_handle<void>
{
    // 17.12.4.2, construct/reset
    constexpr coroutine_handle() noexcept;
    constexpr coroutine_handle(nullptr_t) noexcept;
    coroutine_handle& operator=(nullptr_t) noexcept;

    // 17.12.4.3, export/import
    constexpr void* address() const noexcept;
    static constexpr coroutine_handle from_address(void* addr);

    // 17.12.4.4, observers
    constexpr explicit operator bool() const noexcept;
    bool done() const;

    // 17.12.4.5, resumption
    void operator()() const;
    void resume() const;
    void destroy() const;

    private:
    void* ptr;  // exposition only
};

template<class Promise>
struct coroutine_handle : coroutine_handle<>{
    // 17.12.4.2, construct/reset
    using coroutine_handle<>::coroutine_handle;
    static coroutine_handle from_promise(Promise&);
    coroutine_handle& operator=(nullptr_t) noexcept;

    // 17.12.4.3, export/import
    static constexpr coroutine_handle from_address(void* addr);

    // 17.12.4.6, promise access
    Promise& promise() const;
};

1 An object of type coroutine_handle<T> is called a coroutine handle and can be used to refer to a suspended or executing coroutine. A default-constructed coroutine_handle object does not refer to any coroutine.

2 If a program declares an explicit or partial specialization of coroutine_handle, the behavior is undefined.
17.12.4.2 Construct/reset

constexpr coroutine_handle() noexcept;
constexpr coroutine_handle(nullptr_t) noexcept;

Postconditions: address() == nullptr.

static coroutine_handle from_promise(Promise& p);

Preconditions: p is a reference to a promise object of a coroutine.

Postconditions: addressof(h.promise()) == addressof(p).

Returns: A coroutine handle h referring to the coroutine.

coroutine_handle& operator=(nullptr_t) noexcept;

Postconditions: address() == nullptr.

Returns: *this.

17.12.4.3 Export/import

constexpr void* address() const noexcept;

Returns: ptr.

static constexpr coroutine_handle<> coroutine_handle<>::from_address(void* addr);
static constexpr coroutine_handle<Promise> coroutine_handle<Promise>::from_address(void* addr);

Preconditions: addr was obtained via a prior call to address.

Postconditions: from_address(address()) == *this.

17.12.4.4 Observers

constexpr explicit operator bool() const noexcept;

Returns: address() != nullptr.

bool done() const;

Preconditions: *this refers to a suspended coroutine.

Returns: true if the coroutine is suspended at its final suspend point, otherwise false.

17.12.4.5 Resumption

Resuming a coroutine via resume, operator(), or destroy on an execution agent other than the one on
which it was suspended has implementation-defined behavior unless each execution agent either is an instance
of std::thread or std::jthread, or is the thread that executes main.

[Note 1: A coroutine that is resumed on a different execution agent should avoid relying on consistent thread identity
throughout, such as holding a mutex object across a suspend point. — end note]

[Note 2: A concurrent resumption of the coroutine can result in a data race. — end note]

void operator()() const;
void resume() const;

Preconditions: *this refers to a suspended coroutine. The coroutine is not suspended at its final
suspend point.

Effects: Resumes the execution of the coroutine.

void destroy() const;

Preconditions: *this refers to a suspended coroutine.

Effects: Destroys the coroutine (9.5.4).

17.12.4.6 Promise access

Promise& promise() const;

Preconditions: *this refers to a coroutine.

Returns: A reference to the promise of the coroutine.
17.12.4.7 Comparison operators  

constexpr bool operator==(coroutine_handle<> x, coroutine_handle<> y) noexcept;

Returns: x.address() == y.address().

constexpr strong_ordering operator<=>(coroutine_handle<> x, coroutine_handle<> y) noexcept;

Returns: compare_three_way()(x.address(), y.address()).

17.12.4.8 Hash support

template<class P> struct hash<coroutine_handle<P>>;

The specialization is enabled (20.14.19).

17.12.5 No-op coroutines

17.12.5.1 Class noop_coroutine_promise

struct noop_coroutine_promise {};

The class noop_coroutine_promise defines the promise type for the coroutine referred to by noop_
coroutine_handle (17.12.2).

17.12.5.2 Class coroutine_handle<noop_coroutine_promise>

namespace std {
    template<
    struct coroutine_handle<noop_coroutine_promise> : coroutine_handle<>

    // 17.12.5.2.1, observers
    constexpr explicit operator bool() const noexcept;
    constexpr bool done() const noexcept;

    // 17.12.5.2.2, resumption
    constexpr void operator()() const noexcept;
    constexpr void resume() const noexcept;
    constexpr void destroy() const noexcept;

    // 17.12.5.2.3, promise access
    noop_coroutine_promise& promise() const noexcept;

    // 17.12.5.2.4, address
    constexpr void* address() const noexcept;
    private:
        coroutine_handle<unspecified>;
    }
}

17.12.5.2.1 Observers

constexpr explicit operator bool() const noexcept;

Returns: true.

constexpr bool done() const noexcept;

Returns: false.

17.12.5.2.2 Resumption

constexpr void operator()() const noexcept;
constexpr void resume() const noexcept;
constexpr void destroy() const noexcept;

Effects: None.

Remarks: If noop_coroutine_handle is converted to coroutine_handle<>, calls to operator(),
resume and destroy on that handle will also have no observable effects.

§ 17.12.5.2.2
17.12.5.2.3 Promise access

noop_coroutine.promise& promise() const noexcept;

Returns: A reference to the promise object associated with this coroutine handle.

17.12.5.2.4 Address

constexpr void* address() const noexcept;

Returns: ptr.

Remarks: A noop_coroutine_handle’s ptr is always a non-null pointer value.

17.12.5.3 Function noop_coroutine

noop_coroutine_handle noop_coroutine() noexcept;

Returns: A handle to a coroutine that has no observable effects when resumed or destroyed.

Remarks: A handle returned from noop_coroutine may or may not compare equal to a handle returned from another invocation of noop_coroutine.

17.12.6 Trivial awaitsables

namespace std {
    struct suspend_never {
        constexpr bool await_ready() const noexcept { return true; }
        constexpr void await_suspend(coroutine_handle<> const noexcept {})
        constexpr void await_resume() const noexcept {};
    }
    struct suspend_always {
        constexpr bool await_ready() const noexcept { return false; }
        constexpr void await_suspend(coroutine_handle<> const noexcept {})
        constexpr void await_resume() const noexcept {};
    }
}

[Note 1: The types suspend_never and suspend_always can be used to indicate that an await-expression either never suspends or always suspends, and in either case does not produce a value. — end note]

17.13 Other runtime support

17.13.1 General

Headers <csetjmp> (nonlocal jumps), <csignal> (signal handling), <cstdarg> (variable arguments), and <cstdlib> (runtime environment getenv, system), provide further compatibility with C code.

Calls to the function getenv (17.2.2) shall not introduce a data race (16.4.6.10) provided that nothing modifies the environment.

[Note 1: Calls to the POSIX functions setenv and putenv modify the environment. — end note]

A call to the setlocale function (28.5) may introduce a data race with other calls to the setlocale function or with calls to functions that are affected by the current C locale. The implementation shall behave as if no library function other than locale::global calls the setlocale function.

17.13.2 Header <cstdarg> synopsis

namespace std {
    using va_list = see below;
}

#define va_arg(V, P) see below
#define va_copy(VDST, VSRC) see below
#define va_end(V) see below
#define va_start(V, P) see below

The contents of the header <cstdarg> are the same as the C standard library header <stdarg.h>, with the following changes: The restrictions that ISO C places on the second parameter to the va_start macro in header <stdarg.h> are different in this document. The parameter parmN is the rightmost parameter in the
variable parameter list of the function definition (the one just before the ...). If the parameter \texttt{parmN} is a pack expansion (13.7.4) or an entity resulting from a lambda capture (7.5.5), the program is ill-formed, no diagnostic required. If the parameter \texttt{parmN} is of a reference type, or of a type that is not compatible with the type that results when passing an argument for which there is no parameter, the behavior is undefined.

**See also:** ISO C 7.16.1.1

### 17.13.3 Header \texttt{<csetjmp> synopsis}

```cpp
namespace std {
  using jmp_buf = see below;
  [[noreturn]] void longjmp(jmp_buf env, int val);
}

#define setjmp(env) see below
```

1. The contents of the header \texttt{<csetjmp>} are the same as the C standard library header \texttt{<setjmp.h>},

2. The function signature \texttt{longjmp(jmp_buf jbuf, int val)} has more restricted behavior in this document. A \texttt{setjmp/longjmp} call pair has undefined behavior if replacing the \texttt{setjmp} and \texttt{longjmp} by \texttt{catch} and \texttt{throw} would invoke any non-trivial destructors for any objects with automatic storage duration. A call to \texttt{setjmp} or \texttt{longjmp} has undefined behavior if invoked in a suspension context of a coroutine (7.6.2.4).

**See also:** ISO C 7.13

### 17.13.4 Header \texttt{<csignal> synopsis}

```cpp
namespace std {
  using sig_atomic_t = see below;
  // 17.13.5, signal handlers
  extern "C" using signal-handler = void(int); // exposition only
  signal-handler* signal(int sig, signal-handler* func);
  int raise(int sig);
}

#define SIG_DFL see below
#define SIG_ERR see below
#define SIG_IGN see below
#define SIGABRT see below
#define SIGFPE see below
#define SIGILL see below
#define SIGINT see below
#define SIGSEGV see below
#define SIGTERM see below
```

1. The contents of the header \texttt{<csignal>} are the same as the C standard library header \texttt{<signal.h>},

2. A call to the function \texttt{signal} synchronizes with any resulting invocation of the signal handler so installed.

3. An evaluation is \texttt{signal-safe} unless it includes one of the following:

   (3.1) a call to any standard library function, except for plain lock-free atomic operations and functions explicitly identified as signal-safe;

   (3.2) \texttt{signal}.

---

221) Note that \texttt{va_start} is required to work as specified even if unary \texttt{operator\&} is overloaded for the type of \texttt{parmN}.
[Note 1: This implicitly excludes the use of `new` and `delete` expressions that rely on a library-provided memory allocator. — end note]

(3.2) — an access to an object with thread storage duration;

(3.3) — a `dynamic_cast` expression;

(3.4) — throwing of an exception;

(3.5) — control entering a `try-block` or `function-try-block`;

(3.6) — initialization of a variable with static storage duration requiring dynamic initialization (6.9.3.3, 8.8)*; or

(3.7) — waiting for the completion of the initialization of a variable with static storage duration (8.8).

A signal handler invocation has undefined behavior if it includes an evaluation that is not signal-safe.

* The function `signal` is signal-safe if it is invoked with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler.

**See also:** ISO C 7.14

---

222) Such initialization might occur because it is the first odr-use (6.3) of that variable.
18 Concepts library

18.1 General

This Clause describes library components that C++ programs may use to perform compile-time validation of template arguments and perform function dispatch based on properties of types. The purpose of these concepts is to establish a foundation for equational reasoning in programs.

The following subclauses describe language-related concepts, comparison concepts, object concepts, and callable concepts as summarized in Table 39.

Table 39: Fundamental concepts library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2</td>
<td>Equality preservation</td>
</tr>
<tr>
<td>18.4</td>
<td>Language-related concepts &lt;concepts&gt;</td>
</tr>
<tr>
<td>18.5</td>
<td>Comparison concepts</td>
</tr>
<tr>
<td>18.6</td>
<td>Object concepts</td>
</tr>
<tr>
<td>18.7</td>
<td>Callable concepts</td>
</tr>
</tbody>
</table>

18.2 Equality preservation

An expression is equality-preserving if, given equal inputs, the expression results in equal outputs. The inputs to an expression are the set of the expression’s operands. The output of an expression is the expression’s result and all operands modified by the expression. For the purposes of this subclause, the operands of an expression are the largest subexpressions that include only:

1. An id-expression (7.5.4), and
2. Invocations of the library function templates std::move, std::forward, and std::declval (20.2.4, 20.2.6).

[Example 1: The operands of the expression a = std::move(b) are a and std::move(b). — end example]

Not all input values need be valid for a given expression; e.g., for integers a and b, the expression a / b is not well-defined when b is 0. This does not preclude the expression a / b being equality-preserving. The domain of an expression is the set of input values for which the expression is required to be well-defined.

Expressions required to be equality-preserving are further required to be stable: two evaluations of such an expression with the same input objects are required to have equal outputs absent any explicit intervening modification of those input objects.

[Note 1: This requirement allows generic code to reason about the current values of objects based on knowledge of the prior values as observed via equality-preserving expressions. It effectively forbids spontaneous changes to an object, changes to an object from another thread of execution, changes to an object as side effects of non-modifying expressions, and changes to an object as side effects of modifying a distinct object if those changes could be observable to a library function via an equality-preserving expression that is required to be valid for that object. — end note]

Expressions declared in a requires-expression in the library clauses are required to be equality-preserving, except for those annotated with the comment “not required to be equality-preserving.” An expression so annotated may be equality-preserving, but is not required to be so.

An expression that may alter the value of one or more of its inputs in a manner observable to equality-preserving expressions is said to modify those inputs. The library clauses use a notational convention to specify which expressions declared in a requires-expression modify which inputs: except where otherwise specified, an expression operand that is a non-constant lvalue or rvalue may be modified. Operands that are constant lvalues or rvalues are required to not be modified. For the purposes of this subclause, the cv-qualification and value category of each operand are determined by assuming that each template type parameter denotes a cv-qualified complete non-array object type.

Where a requires-expression declares an expression that is non-modifying for some constant lvalue operand, additional variations of that expression that accept a non-constant lvalue or (possibly constant) rvalue for
the given operand are also required except where such an expression variation is explicitly required with differing semantics. These *implicit expression variations* are required to meet the semantic requirements of the declared expression. The extent to which an implementation validates the syntax of the variations is unspecified.

[Example 2:]

```cpp
template<class T> concept C = requires(T a, T b, const T c, const T d) {
  c == d; // #1
  a = std::move(b); // #2
  a = c; // #3
};
```

For the above example:

1. Expression #1 does not modify either of its operands, #2 modifies both of its operands, and #3 modifies only its first operand `a`.
2. Expression #1 implicitly requires additional expression variations that meet the requirements for `c == d` (including non-modification), as if the expressions

   ```cpp
   c == std::move(d);
   std::move(c) == d;
   std::move(c) == std::move(b);
   ```

   had been declared as well.

3. Expression #3 implicitly requires additional expression variations that meet the requirements for `a == c` (including non-modification of the second operand), as if the expressions `a == b` and `a = std::move(c)` had been declared. Expression #3 does not implicitly require an expression variation with a non-constant rvalue second operand, since expression #2 already specifies exactly such an expression explicitly.

   —end example]

8 [Example 3: The following type `T` meets the explicitly stated syntactic requirements of concept `C` above but does not meet the additional implicit requirements:

```cpp
struct T {
  bool operator==(const T&) const { return true; }
  bool operator==(T&) = delete;
};
```

`T` fails to meet the implicit requirements of `C`, so `T` satisfies but does not model `C`. Since implementations are not required to validate the syntax of implicit requirements, it is unspecified whether an implementation diagnoses as ill-formed a program that requires `C<T>`. —end example]

### 18.3 Header `<concepts>` synopsis

```cpp
namespace std {
  // 18.4, language-related concepts
  // 18.4.2, concept same_as
  template<class T, class U>
  concept same_as = see below;

  // 18.4.3, concept derived_from
  template<class Derived, class Base>
  concept derived_from = see below;

  // 18.4.4, concept convertible_to
  template<class From, class To>
  concept convertible_to = see below;

  // 18.4.5, concept common_reference_with
  template<class T, class U>
  concept common_reference_with = see below;
}
```

§ 18.3 554
// 18.4.6, concept common_with
template<class T, class U>
concept common_with = see below;

// 18.4.7, arithmetic concepts
template<class T>
concept integral = see below;
template<class T>
concept signed_integral = see below;
template<class T>
concept unsigned_integral = see below;
template<class T>
concept floating_point = see below;

// 18.4.8, concept assignable_from
template<class LHS, class RHS>
concept assignable_from = see below;

// 18.4.9, concept swappable
namespace ranges {
    inline namespace unspecified {
        inline constexpr unspecified swap = unspecified;
    }
}
template<class T>
concept swappable = see below;
template<class T, class U>
concept swappable_with = see below;

// 18.4.10, concept destructible
template<class T>
concept destructible = see below;

// 18.4.11, concept constructible_from
template<class T, class... Args>
concept constructible_from = see below;

// 18.4.12, concept default_initializable
template<class T>
concept default_initializable = see below;

// 18.4.13, concept move_constructible
template<class T>
concept move_constructible = see below;

// 18.4.14, concept copy_constructible
template<class T>
concept copy_constructible = see below;

// 18.5, comparison concepts
// 18.5.3, concept equality_comparable
template<class T>
concept equality_comparable = see below;
template<class T, class U>
concept equality_comparable_with = see below;

// 18.5.4, concept totally_ordered
template<class T>
concept totally_ordered = see below;
template<class T, class U>
concept totally_ordered_with = see below;
18.4 Language-related concepts

18.4.1 General

Subclause 18.4 contains the definition of concepts corresponding to language features. These concepts express relationships between types, type classifications, and fundamental type properties.

18.4.2 Concept same_as

template<class T, class U>
concept same_as_impl = is_same_v<T, U>; // exposition only

template<class T, class U>
concept same_as = same_as_impl<T, U> && same_as_impl<U, T>;

[Note 1: same_as<T, U> subsumes same_as<U, T> and vice versa. — end note]

18.4.3 Concept derived_from

template<class Derived, class Base>
concept derived_from =
is_base_of_v<Base, Derived> &&
is_convertible_v<const volatile Derived*, const volatile Base*>;

[Note 1: derived_from<Derived, Base> is satisfied if and only if Derived is publicly and unambiguously derived from Base, or Derived and Base are the same class type ignoring cv-qualifiers. — end note]
18.4.4 Concept convertible_to

Given types From and To and an expression E such that decltype((E)) is add_rvalue_reference_t<From>, convertible_to<From, To> requires E to be both implicitly and explicitly convertible to type To. The implicit and explicit conversions are required to produce equal results.

```cpp
template<class From, class To>
concept convertible_to =
    is_convertible_v<From, To> &&
    requires(add_rvalue_reference_t<From> (&f)()) {
    static_cast<To>(f());
    };
```

Let FromR be add_rvalue_reference_t<From> and test be the invented function:

```cpp
to test(FromR (&f)()) {
    return f();
}
```
and let f be a function with no arguments and return type FromR such that f() is equality-preserving. Types From and To model convertible_to<From, To> only if:

1. To is not an object or reference-to-object type, or static_cast<To>(f()) is equal to test(f).
2. FromR is not a reference-to-object type, or
   1. If FromR is an rvalue reference to a non const-qualified type, the resulting state of the object referenced by f() after either above expression is valid but unspecified (16.4.6.16).
   2. Otherwise, the object referred to by f() is not modified by either above expression.

18.4.5 Concept common_reference_with

For two types T and U, if common_reference_t<T, U> is well-formed and denotes a type C such that both convertible_to<T, C> and convertible_to<U, C> are modeled, then T and U share a common reference type, C.

[Note 1: C could be the same as T, or U, or it could be a different type. C can be a reference type. — end note]

```cpp
template<class T, class U>
concept common_reference_with =
    same_as<common_reference_t<T, U>, common_reference_t<U, T>> &&
    convertible_to<T, common_reference_t<T, U>> &&
    convertible_to<U, common_reference_t<T, U>>;
```

Let C be common_reference_t<T, U>. Let t1 and t2 be equality-preserving expressions (18.2) such that decltype((t1)) and decltype((t2)) are each T, and let u1 and u2 be equality-preserving expressions such that decltype((u1)) and decltype((u2)) are each U. T and U model common_reference_with<T, U> only if:

1. C(t1) equals C(t2) if and only if t1 equals t2, and
2. C(u1) equals C(u2) if and only if u1 equals u2.

[Note 2: Users can customize the behavior of common_reference_with by specializing the basic_common_reference class template (20.15.8.7). — end note]

18.4.6 Concept common_with

If T and U can both be explicitly converted to some third type, C, then T and U share a common type, C.

[Note 1: C could be the same as T, or U, or it could be a different type. C might not be unique. — end note]

```cpp
template<class T, class U>
concept common_with =
    same_as<common_type_t<T, U>, common_type_t<U, T>> &&
    requires {
    static_cast<common_type_t<T, U>>(declval<T>());
    static_cast<common_type_t<T, U>>(declval<U>());
    } &&
```
common_reference_with<
  add_lvalue_reference_t<const T>,
add_lvalue_reference_t<const U>> &&
common_reference_with<
  add_lvalue_reference_t<common_type_t<T, U>>,
  common_reference_t<
    add_lvalue_reference_t<const T>,
    add_lvalue_reference_t<const U>>>;

Let $C$ be $\text{common_type_t<T, U>}$. Let $t_1$ and $t_2$ be equality-preserving expressions (18.2) such that $\text{decltype((t_1))}$ and $\text{decltype((t_2))}$ are each $T$, and let $u_1$ and $u_2$ be equality-preserving expressions such that $\text{decltype((u_1))}$ and $\text{decltype((u_2))}$ are each $U$. $T$ and $U$ model $\text{common_with<T, U>}$ only if:

(2.1) $C(t_1)$ equals $C(t_2)$ if and only if $t_1$ equals $t_2$, and
(2.2) $C(u_1)$ equals $C(u_2)$ if and only if $u_1$ equals $u_2$.

[Note 2: Users can customize the behavior of $\text{common_with}$ by specializing the $\text{common_type}$ class template (20.15.8.7). —end note]

### 18.4.7 Arithmetic concepts

[concepts.arithmetic]

```cpp
template<class T>
concept integral = is_integral_v<T>;

template<class T>
concept signed_integral = integral<T> && is_signed_v<T>;

template<class T>
concept unsigned_integral = integral<T> && !signed_integral<T>;

template<class T>
concept floating_point = is_floating_point_v<T>;
```

[Note 1: $\text{signed_integral}$ can be modeled even by types that are not signed integer types (6.8.2); for example, char. —end note]

[Note 2: $\text{unsigned_integral}$ can be modeled even by types that are not unsigned integer types (6.8.2); for example, bool. —end note]

### 18.4.8 Concept assignable_from

[concept.assignable]

```cpp
template<class LHS, class RHS>
concept assignable_from =
  is_lvalue_reference_v<LHS> &&
common_reference_with<
  const remove_reference_t<LHS>,
  const remove_reference_t<RHS>> &&
requires(LHS lhs, RHS&& rhs) {
  (lhs = std::forward<RHS>(rhs)) -> same_as<LHS>;
};
```

Let:

(1.1) $l$ be an lvalue that refers to an object $l$copy such that $\text{decltype((l))}$ is $LHS$,
(1.2) $r$ be an expression such that $\text{decltype((r))}$ is $RHS$, and
(1.3) $rcopy$ be a distinct object that is equal to $r$.

$LHS$ and $RHS$ model $\text{assignable_from<LHS, RHS>}$ only if

(1.4) $\text{addressof}(l = r) == \text{addressof}(l$copy$)$.
(1.5) After evaluating $l = r$:

(1.5.1) $l$ is equal to $rcopy$, unless $r$ is a non-const xvalue that refers to $l$copy.
(1.5.2) If $r$ is a non-const xvalue, the resulting state of the object to which it refers is valid but unspecified (16.4.6.16).
(1.5.3) Otherwise, if $r$ is a glvalue, the object to which it refers is not modified.

[Note 1: Assignment need not be a total function (16.3.2.3); in particular, if assignment to an object $x$ can result in a modification of some other object $y$, then $x = y$ is likely not in the domain of $=$. —end note]
18.4.9 Concept swappable

Let $t_1$ and $t_2$ be equality-preserving expressions that denote distinct equal objects of type $T$, and let $u_1$ and $u_2$ similarly denote distinct equal objects of type $U$.

[Note 1: $t_1$ and $u_1$ can denote distinct objects, or the same object. — end note]

An operation exchanges the values denoted by $t_1$ and $u_1$ if and only if the operation modifies neither $t_2$ nor $u_2$ and:

1. If $T$ and $U$ are the same type, the result of the operation is that $t_1$ equals $u_2$, and $u_1$ equals $t_2$.
2. If $T$ and $U$ are different types and common_reference_with<decltype($(t_1)$), decltype($(u_1)$)> is modeled, the result of the operation is that $C(t_1)$ equals $C(u_2)$ and $C(u_1)$ equals $C(t_2)$ where $C$ is common_reference_t<decltype($(t_1)$), decltype($(u_1)$)>.

The name ranges::swap denotes a customization point object (16.3.3.3.6). The expression ranges::swap($E_1$, $E_2$) for subexpressions $E_1$ and $E_2$ is expression-equivalent to an expression $S$ determined as follows:

1. $S$ is (void)swap($E_1$, $E_2$) if $E_1$ or $E_2$ has class or enumeration type (6.8.3) and that expression is valid, with overload resolution performed in a context that includes the declaration
   
   template<class T>
   void swap(T&, T&) = delete;

   and does not include a declaration of ranges::swap. If the function selected by overload resolution does not exchange the values denoted by $E_1$ and $E_2$, the program is ill-formed, no diagnostic required.
2. Otherwise, if $E_1$ and $E_2$ are lvalues of array types (6.8.3) with equal extent and ranges::swap(*$E_1$, *$E_2$) is a valid expression, $S$ is (void)ranges::swap_ranges($E_1$, $E_2$), except that noexcept($S$) is equal to noexcept(ranges::swap(*$E_1$, *$E_2$)).
3. Otherwise, if $E_1$ and $E_2$ are lvalues of the same type $T$ that models move_constructible<T> and assignable_from<T&, T>, $S$ is an expression that exchanges the denoted values. $S$ is a constant expression if
   
   (2.3.1) $T$ is a literal type (6.8),
   
   (2.3.2) both $E_1 = std::move(E_2)$ and $E_2 = std::move(E_1)$ are constant subexpressions (3.14), and
   
   (2.3.3) the full-expressions of the initializers in the declarations
       
       $T$ t1(std::move(E1));
       $T$ t2(std::move(E2));

   are constant subexpressions.

   noexcept($S$) is equal to is_nothrow_move_constructible_v<T> && is_nothrow_move_assignable_v<T>.

   (2.4) Otherwise, ranges::swap($E_1$, $E_2$) is ill-formed.

   [Note 2: This case can result in substitution failure when ranges::swap($E_1$, $E_2$) appears in the immediate context of a template instantiation. — end note]

3. Whenever ranges::swap($E_1$, $E_2$) is a valid expression, it exchanges the values denoted by $E_1$ and $E_2$ and has type void. — end note]

   template<class T>
   concept swappable = requires(T& a, T& b) { ranges::swap(a, b); };

   template<class T, class U>
   concept swappable_with =
       common_reference_with<T, U> &&
       requires(T&& t, U&& u) {
           ranges::swap(std::forward<T>(t), std::forward<T>(t));
           ranges::swap(std::forward<U>(u), std::forward<U>(u));
           ranges::swap(std::forward<T>(t), std::forward<U>(u));
           ranges::swap(std::forward<U>(u), std::forward<T>(t));
       }

   [Note 4: The semantics of the swappable and swappable_with concepts are fully defined by the ranges::swap customization point object. — end note]

223) The name swap is used here unqualified.
Example 1: User code can ensure that the evaluation of `swap` calls is performed in an appropriate context under the various conditions as follows:

```cpp
#include <cassert>
#include <concepts>
#include <utility>
namespace ranges = std::ranges;

template<class T, std::swappable_with<T> U>
void value_swap(T&& t, U&& u) {
    ranges::swap(std::forward<T>(t), std::forward<U>(u));
}

template<std::swappable T>
void lv_swap(T& t1, T& t2) {
    ranges::swap(t1, t2);
}

namespace N {
    struct A { int m; };
    struct Proxy {
        A* a;
        Proxy(A& a) : a(&a) {}  // exposition only
        friend void swap(Proxy x, Proxy y) {
            ranges::swap(*x.a, *y.a);
        }
    };
    Proxy proxy(A& a) { return Proxy{ a }; }
}

int main() {
    int i = 1, j = 2;
    lv_swap(i, j);
    assert(i == 2 && j == 1);

    N::A a1 = { 5 }, a2 = { -5 };
    value_swap(a1, proxy(a2));
    assert(a1.m == -5 && a2.m == 5);
}
```

18.4.10 Concept destructible

The `destructible` concept specifies properties of all types, instances of which can be destroyed at the end of their lifetime, or reference types.

```cpp
template<class T>
concept destructible = is_nothrow_destructible_v<T>;
```

[Note 1: Unlike the Cpp17Destructible requirements (Table 32), this concept forbids destructors that are potentially throwing, even if a particular invocation of the destructor does not actually throw. —end note]

18.4.11 Concept constructible_from

The `constructible_from` concept constrains the initialization of a variable of a given type with a particular set of argument types.

```cpp
template<class T, class... Args>
concept constructible_from = destructible<T> && is_constructible_v<T, Args...>;
```

18.4.12 Concept default_initializable

```cpp
template<class T>
inline constexpr bool is-default-initializable = see below; // exposition only
```

§ 18.4.12
18.4.13 Concept move_constructible

```cpp
template<class T>
concept move_constructible = constructible_from<T, T> && convertible_to<T, T>;
```

1 For a type $T$, $\text{move\_constructible}<T>$ is true if and only if the variable definition
$T t$;

is well-formed for some invented variable $t$; otherwise it is false. Access checking is performed as if in
a context unrelated to $T$. Only the validity of the immediate context of the variable initialization is
considered.

18.4.14 Concept copy_constructible

```cpp
template<class T>
concept copy_constructible =
  move_constructible<T> &&
  constructible_from<T, T&> && convertible_to<T&, T> &&
  constructible_from<T, const T&> && convertible_to<const T&, T> &&
  constructible_from<T, const T> && convertible_to<const T, T>;
```

1 If $T$ is an object type, then let $rv$ be an rvalue of type $T$ and $u2$ a distinct object of type $T$ equal to $rv$.
$T$ models $\text{move\_constructible}$ only if

1. After the definition $T u = rv$; $u$ is equal to $u2$.
2. $T(rv)$ is equal to $u2$.
3. If $T$ is not $\text{const}$, $rv$'s resulting state is valid but unspecified (16.4.6.16); otherwise, it is unchanged.

18.5 Comparison concepts

18.5.2 Boolean testability

```cpp
template<class T>
concept boolean-testable-impl = convertible_to<T, bool>; // exposition only
```

2 Let $e$ be an expression such that $\text{decltype((e))}$ is $T$. $T$ models $\text{boolean\_testable-impl}$ only if:

1. either $\text{remove_cvref}_T(e)$ is not a class type, or name lookup for the names $\text{operator&&}$ and $\text{operator||}$ within the scope of $\text{remove_cvref}_T(e)$ as if by class member access lookup (11.8) results in an empty declaration set; and
2. name lookup for the names $\text{operator&&}$ and $\text{operator||}$ in the associated namespaces and entities of
$T$ (6.5.3) finds no disqualifying declaration (defined below).

3 A disqualifying parameter is a function parameter whose declared type $P$

1. is not dependent on a template parameter, and there exists an implicit conversion sequence (12.4.4.2) from $e$ to $P$; or

§ 18.5.2 561
— is dependent on one or more template parameters, and either

— \( P \) contains no template parameter that participates in template argument deduction (13.10.3.6), or

— template argument deduction using the rules for deducing template arguments in a function call (13.10.3.2) and \( e \) as the argument succeeds.

4 A **key parameter** of a function template \( D \) is a function parameter of type \( cv X \) or reference thereto, where \( X \) names a specialization of a class template that is a member of the same namespace as \( D \), and \( X \) contains at least one template parameter that participates in template argument deduction.

[Example 1: In

```cpp
namespace Z {
  template<class> struct C {};
  template<class T>
    void operator&&(C<T> x, T y);
  template<class T>
    void operator||(C<type_identity_t<T>> x, T y);
}
```

the declaration of \( Z::operator&& \) contains one key parameter, \( C<T> x \), and the declaration of \( Z::operator|| \) contains no key parameters. —end example]

5 A **disqualifying declaration** is

— a (non-template) function declaration that contains at least one disqualifying parameter; or

— a function template declaration that contains at least one disqualifying parameter, where

— at least one disqualifying parameter is a key parameter; or

— the declaration contains no key parameters; or

— the declaration declares a function template that is not visible in its namespace (9.8.2.3).

[Note 1: The intention is to ensure that given two types \( T1 \) and \( T2 \) that each model \( boolean\text{-}testable\text{-}impl \), the \&\& and || operators within the expressions \( \text{declval<T1>() && declval<T2>()} \) and \( \text{declval<T1>() || declval<T2>()} \) resolve to the corresponding built-in operators. —end note]

```cpp
template<class T>
concept boolean\text{-}testable = // exposition only
  boolean\text{-}testable\text{-}impl<T> \&\& requires (T\& t) {
    \{ !std::forward<T>(t) \} \rightarrow boolean\text{-}testable\text{-}impl;
  }
```

7 Let \( e \) be an expression such that \( \text{decltype((e))} \) is \( T \). \( T \) models \( boolean\text{-}testable \) only if \( \text{bool(e)} = !\text{bool(!e)} \).

8 [Example 2: The types \( bool \), \( true\text{-}type \) (20.15.3), \( int* \), and \( bitset<N>::reference \) (20.9.2) model \( boolean\text{-}testable \). —end example]

### 18.5.3 Concept equality\_comparable

```cpp
template<class T, class U>
concept weakly\_equality\_comparable\_with = // exposition only
  requires(const remove\_reference\_t<T>& t,
           const remove\_reference\_t<U>& u) {
    \{ t == u \} \rightarrow boolean\_testable;
    \{ t != u \} \rightarrow boolean\_testable;
    \{ u == t \} \rightarrow boolean\_testable;
    \{ u != t \} \rightarrow boolean\_testable;
  }
```

1 Given types \( T \) and \( U \), let \( t \) and \( u \) be lvalues of types \( \text{const remove\_reference\_t<T>} \) and \( \text{const remove\_reference\_t<U>} \) respectively. \( T \) and \( U \) model \( weakly\text{-}equality\text{-}comparable\_with<T,U> \) only if

— \( t == u, u == t, t != u, \) and \( u != t \) have the same domain.

— \( bool(u == t) == bool(t == u) \).

— \( bool(t != u) == !bool(t == u) \).

— \( bool(u != t) == bool(t != u) \).
template<class T>
  concept equality_comparable = weakly-equality-comparable-with<T, T>;

2. Let a and b be objects of type T. T models equality_comparable only if bool(a == b) is true when a is equal to b (18.2), and false otherwise.

[Note 1: The requirement that the expression a == b is equality-preserving implies that == is transitive and symmetric. — end note]

template<class T, class U>
  concept equality_comparable_with =
    equality_comparable<T> && equality_comparable<U> &&
    common_reference_with<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
    equality_comparable<
      common_reference_t<
        const remove_reference_t<T>&,
        const remove_reference_t<U>&>> &&
    weakly-equality-comparable-with<T, U>;

4. Given types T and U, let t be an lvalue of type const remove_reference_t<T>, u be an lvalue of type const remove_reference_t<U>, and C be:

  common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>

T and U model equality_comparable_with<T, U> only if bool(t == u) == bool(C(t) == C(u)).

18.5.4 Concept totally_ordered

[concept.totallyordered]

template<class T>
  concept totally_ordered =
    equality_comparable<T> && partially-ordered-with<T, T>;

1. Given a type T, let a, b, and c be lvalues of type const remove_reference_t<T>. T models totally_ordered only if

(1.1) Exactly one of bool(a < b), bool(a > b), or bool(a == b) is true.

(1.2) If bool(a < b) and bool(b < c), then bool(a < c).

(1.3) bool(a <= b) == !bool(b < a).

(1.4) bool(a >= b) == !bool(a < b).

template<class T, class U>
  concept totally_ordered_with =
    totally_ordered<T> && totally_ordered<U> &&
    equality_comparable_with<T, U> &&
    totally_ordered<
      common_reference_t<
        const remove_reference_t<T>&,
        const remove_reference_t<U>&>> &&
    partially-ordered-with<T, U>;

2. Given types T and U, let t be an lvalue of type const remove_reference_t<T>, u be an lvalue of type const remove_reference_t<U>, and C be:

  common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>

T and U model totally_ordered_with<T, U> only if

(2.1) bool(t < u) == bool(C(t) < C(u)).

(2.2) bool(t > u) == bool(C(t) > C(u)).

(2.3) bool(t <= u) == bool(C(t) <= C(u)).

(2.4) bool(t >= u) == bool(C(t) >= C(u)).

(2.5) bool(u < t) == bool(C(u) < C(t)).

(2.6) bool(u > t) == bool(C(u) > C(t)).

(2.7) bool(u <= t) == bool(C(u) <= C(t)).

(2.8) bool(u >= t) == bool(C(u) >= C(t)).
18.6 Object concepts  
This subclause describes concepts that specify the basis of the value-oriented programming style on which the library is based.

```cpp
1 template<class T>
   concept movable = is_object<T> && move_constructible<T> &&
                        assignable_from<T, T> && swappable<T>;

2 template<class T>
   concept copyable = copy_constructible<T> && movable<T> && assignable_from<T, T> &&
                        assignable_from<T, const T> && assignable_from<T, const T>;

3 template<class T>
   concept semiregular = copyable<T> && default_initializable<T>;

4 template<class T>
   concept regular = semiregular<T> && equality_comparable<T>;
```

18.7 Callable concepts  

18.7.1 General  

1 The concepts in subclause 18.7 describe the requirements on function objects (20.14) and their arguments.

18.7.2 Concept invocable  

1 The `invocable` concept specifies a relationship between a callable type (20.14.3) \( F \) and a set of argument types \( \text{Args...} \) which can be evaluated by the library function `invoke` (20.14.5).

```cpp
2 template<class F, class... Args>
   concept invocable = requires(F&& f, Args&&... args) {
      invoke(std::forward<F>(f), std::forward<Args>(args)...); // not required to be equality-preserving
   };
```

18.7.3 Concept regular_invocable  

1 The `regular_invocable` concept specifies that the `invoke` function call expression shall be equality-preserving (18.2) and shall not modify the function object or the arguments.

```cpp
2 template<class F, class... Args>
   concept regular_invocable = invocable<F, Args...>;
```

18.7.4 Concept predicate  

```cpp
1 template<class F, class... Args>
   concept predicate =
      regular_invocable<F, Args...> && boolean-testable<invoke_result_t<F, Args...>>;
```

18.7.5 Concept relation  

```cpp
template<class R, class T, class U>
   concept relation =
      predicate<R, T, T> && predicate<R, U, U> &&
      predicate<R, T, U> && predicate<R, U, T>;
```
18.7.6 Concept equivalence_relation

```cpp
template<class R, class T, class U>
concept equivalence_relation = relation<R, T, U>;
```

A relation models equivalence_relation only if it imposes an equivalence relation on its arguments.

18.7.7 Concept strict_weak_order

```cpp
template<class R, class T, class U>
concept strict_weak_order = relation<R, T, U>;
```

A relation models strict_weak_order only if it imposes a strict weak ordering on its arguments.

The term strict refers to the requirement of an irreflexive relation (!comp(x, x) for all x), and the term weak to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define equiv(a, b) as !comp(a, b) && !comp(b, a), then the requirements are that comp and equiv both be transitive relations:

1. comp(a, b) && comp(b, c) implies comp(a, c)
2. equiv(a, b) && equiv(b, c) implies equiv(a, c)

[Note 1: Under these conditions, it can be shown that
1. equiv is an equivalence relation,
2. comp induces a well-defined relation on the equivalence classes determined by equiv, and
3. the induced relation is a strict total ordering.
— end note]
19 Diagnostics library

19.1 General

This Clause describes components that C++ programs may use to detect and report error conditions.

The following subclauses describe components for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes, as summarized in Table 40.

Table 40: Diagnostics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.2</td>
<td>&lt;stdexcept&gt;</td>
</tr>
<tr>
<td>19.3</td>
<td>&lt;cassert&gt;</td>
</tr>
<tr>
<td>19.4</td>
<td>&lt;cerrno&gt;</td>
</tr>
<tr>
<td>19.5</td>
<td>&lt;system_error&gt;</td>
</tr>
</tbody>
</table>

19.2 Exception classes

19.2.1 General

The C++ standard library provides classes to be used to report certain errors in C++ programs. In the error model reflected in these classes, errors are divided into two broad categories: logic errors and runtime errors.

The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program. In theory, they are preventable.

By contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The header <stdexcept> defines several types of predefined exceptions for reporting errors in a C++ program. These exceptions are related by inheritance.

19.2.2 Header <stdexcept> synopsis

namespace std {
    class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
    class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
}

19.2.3 Class logic_error

namespace std {
    class logic_error : public exception {
        public:
            explicit logic_error(const string& what_arg);
            explicit logic_error(const char* what_arg);
    };
}

The class logic_error defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

1 The class logic_error defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

logic_error(const string& what_arg);

Postconditions: strcmp(what(), what_arg.c_str()) == 0.
logic_error(const char* what_arg);

Postconditions: strcmp(what(), what_arg) == 0.

19.2.4 Class domain_error

namespace std {
    class domain_error : public logic_error {
        public:
            explicit domain_error(const string& what_arg);
            explicit domain_error(const char* what_arg);
    };
}

1 The class domain_error defines the type of objects thrown as exceptions by the implementation to report domain errors.

domain_error(const string& what_arg);

Postconditions: strcmp(what(), what_arg.c_str()) == 0.

domain_error(const char* what_arg);

Postconditions: strcmp(what(), what_arg) == 0.

19.2.5 Class invalid_argument

namespace std {
    class invalid_argument : public logic_error {
        public:
            explicit invalid_argument(const string& what_arg);
            explicit invalid_argument(const char* what_arg);
    };
}

1 The class invalid_argument defines the type of objects thrown as exceptions to report an invalid argument.

invalid_argument(const string& what_arg);

Postconditions: strcmp(what(), what_arg.c_str()) == 0.

invalid_argument(const char* what_arg);

Postconditions: strcmp(what(), what_arg) == 0.

19.2.6 Class length_error

namespace std {
    class length_error : public logic_error {
        public:
            explicit length_error(const string& what_arg);
            explicit length_error(const char* what_arg);
    };
}

1 The class length_error defines the type of objects thrown as exceptions to report an attempt to produce an object whose length exceeds its maximum allowable size.

length_error(const string& what_arg);

Postconditions: strcmp(what(), what_arg.c_str()) == 0.

length_error(const char* what_arg);

Postconditions: strcmp(what(), what_arg) == 0.

19.2.7 Class out_of_range

namespace std {
    class out_of_range : public logic_error {
        public:
            explicit out_of_range(const string& what_arg);
            explicit out_of_range(const char* what_arg);
    };
}
The class `out_of_range` defines the type of objects thrown as exceptions to report an argument value not in its expected range.

```cpp
out_of_range(const string& what_arg);
```  
\textit{Postconditions:} `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
out_of_range(const char* what_arg);
```  
\textit{Postconditions:} `strcmp(what(), what_arg) == 0`.

### 19.2.8 Class `runtime_error`

```
namespace std {
    class runtime_error : public exception {
        public:
            explicit runtime_error(const string& what_arg);
            explicit runtime_error(const char* what_arg);
    };
}
```

The class `runtime_error` defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.

```cpp
runtime_error(const string& what_arg);
```  
\textit{Postconditions:} `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
runtime_error(const char* what_arg);
```  
\textit{Postconditions:} `strcmp(what(), what_arg) == 0`.

### 19.2.9 Class `range_error`

```
namespace std {
    class range_error : public runtime_error {
        public:
            explicit range_error(const string& what_arg);
            explicit range_error(const char* what_arg);
    };
}
```

The class `range_error` defines the type of objects thrown as exceptions to report range errors in internal computations.

```cpp
range_error(const string& what_arg);
```  
\textit{Postconditions:} `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
range_error(const char* what_arg);
```  
\textit{Postconditions:} `strcmp(what(), what_arg) == 0`.

### 19.2.10 Class `overflow_error`

```
namespace std {
    class overflow_error : public runtime_error {
        public:
            explicit overflow_error(const string& what_arg);
            explicit overflow_error(const char* what_arg);
    };
}
```

The class `overflow_error` defines the type of objects thrown as exceptions to report an arithmetic overflow error.

```cpp
overflow_error(const string& what_arg);
```  
\textit{Postconditions:} `strcmp(what(), what_arg.c_str()) == 0`. 

\section*{§ 19.2.10}
overflow_error(const char* what_arg);

Postconditions: strcmp(what(), what_arg) == 0.

19.2.11 Class underflow_error

namespace std {
    class underflow_error : public runtime_error {
        public:
            explicit underflow_error(const string& what_arg);
            explicit underflow_error(const char* what_arg);
    };
}

1 The class underflow_error defines the type of objects thrown as exceptions to report an arithmetic underflow error.

underflow_error(const string& what_arg);

Postconditions: strcmp(what(), what_arg.c_str()) == 0.

underflow_error(const char* what_arg);

Postconditions: strcmp(what(), what_arg) == 0.

19.3 Assertions

19.3.1 General

1 The header <cassert> provides a macro for documenting C++ program assertions and a mechanism for disabling the assertion checks.

19.3.2 Header <cassert> synopsis

#define assert(E) see below

1 The contents are the same as the C standard library header <assert.h>, except that a macro named static_assert is not defined.

See also: ISO C 7.2

19.3.3 The assert macro

An expression assert(E) is a constant subexpression (3.14), if

(1.1) — NDEBUG is defined at the point where assert is last defined or redefined, or

(1.2) — E contextually converted to bool (7.3) is a constant subexpression that evaluates to the value true.

19.4 Error numbers

19.4.1 General

1 The contents of the header <cerrno> are the same as the POSIX header <errno.h>, except that errno shall be defined as a macro.

[Note 1: The intent is to remain in close alignment with the POSIX standard. — end note]

A separate errno value shall be provided for each thread.

19.4.2 Header <cerrno> synopsis

#define errno see below

#define E2BIG see below
#define EACCES see below
#define EADDRINUSE see below
#define EADDRNOTAVAIL see below
#define EAFNOSUPPORT see below
#define EAGAIN see below
#define EALREADY see below
#define EBADF see below
#define EBADMSG see below
#define EBUSY see below
#define ECANCELED see below
#define ECHILD see below
#define ECONNABORTED see below
#define ECONNREFUSED see below
#define ECONNRESET see below
#define EDEADLK see below
#define EDESTADDRREQ see below
#define EDOM see below
#define EEXIST see below
#define EFAULT see below
#define EFBIG see below
#define EHOSTUNREACH see below
#define EIDRM see below
#define EILSEQ see below
#define EINPROGRESS see below
#define EINTR see below
#define EINVAL see below
#define EIO see below
#define EISCONN see below
#define EISDIR see below
#define ELOOP see below
#define EMFILE see below
#define EMLINK see below
#define EMSGSIZE see below
#define ENAMETOOLONG see below
#define ENETDOWN see below
#define ENETRESET see below
#define ENETUNREACH see below
#define ENFILE see below
#define ENOBUFS see below
#define ENODATA see below
#define ENODEV see below
#define ENOENT see below
#define ENOEXEC see below
#define ENOLCK see below
#define ENOLINK see below
#define ENOMEM see below
#define ENOMSG see below
#define ENOPROTOOPT see below
#define ENOSPC see below
#define ENOSR see below
#define ENOSTR see below
#define ENOSYS see below
#define ENOTCONN see below
#define ENOTDIR see below
#define ENOTEMPTY see below
#define ENOTRECOVERABLE see below
#define ENOTSOCK see below
#define ENOTSUP see below
#define ENOTTY see below
#define ENXIO see below
#define EOPNOTSUPP see below
#define EOVERFLOW see below
#define EOWNERDEAD see below
#define EPERM see below
#define EPIPE see below
#define EPROTO see below
#define EPROTONOSUPPORT see below
#define EPROTOTYPE see below
#define ERANGE see below
#define ESPIPE see below
#define ESRCH see below
#define ETIME see below
#define ENOTRECOVERABLE see below
#define ENOTSUP see below
#define ERANGE see below
#define EROFS see below
#define ESPIPE see below
#define ESRCH see below
#define ETIME see below
The meaning of the macros in this header is defined by the POSIX standard.

See also: ISO C 7.5

19.5 System error support

19.5.1 General

Subclause 19.5 describes components that the standard library and C++ programs may use to report error conditions originating from the operating system or other low-level application program interfaces.

2 Components described in 19.5 shall not change the value of errno (19.4). Implementations should leave the error states provided by other libraries unchanged.

19.5.2 Header <system_error> synopsis

```cpp
#include <compare>  // see 17.11.1

namespace std {
    class error_category;
    const error_category& generic_category() noexcept;
    const error_category& system_category() noexcept;

    class error_code;
    class error_condition;
    class system_error;

    template<class T>
    struct is_error_code_enum : public false_type {};

    template<class T>
    struct is_error_condition_enum : public false_type {};

    enum class errc {
        address_family_not_supported,           // EAFNOSUPPORT
        address_in_use,                         // EADDRINUSE
        address_not_available,                  // EADDRNOTAVAIL
        already_connected,                      // EISCONN
        argument_list_too_long,                 // E2BIG
        argument_out_of_domain,                 // EDOM
        bad_address,                            //EFAULT
        bad_file_descriptor,                    //EBADF
        bad_message,                            //EBADMSG
        broken_pipe,                            //EPipe
        connection_aborted,                     //ECONNABORTED
        connection_already_in_progress,         //EALREADY
        connection_refused,                     //ECONNREFUSED
        connection_reset,                       //ECONNRESET
        cross_device_link,                      //EXDEV
        destination_address_required,           //EDESTADDRREQ
        device_or_resource_busy,                //EBUSY
        directory_not_empty,                    //ENOTEMPTY
        executable_format_error,                //ENXEXEC
        file_exists,                            //EXIST
        file_too_large,                          //EFBIG
        filename_too_long,                      //ENAMETOOLONG
        function_not_supported,                 //ENOSYS
        host_unreachable,                       //EHOSTUNREACH
        identifier_removed,                     //EIDRM
        illegal_byte_sequence,                  //EILSEQ
        inappropriate_io_control_operation,     //ENOTTY
        interrupted,                            //EINTR
    }
```
invalid_argument,    // EINVAL
invalid_seek,      // EPIPE
io_error,         // EIO
is_a_directory,   // EISDIR
message_size,     // EMSGSIZE
network_down,     // ENETDOWN
network_reset,    // ENETRESET
network_unreachable, // ENETUNREACH
no_buffer_space,  // ENOBUFS
no_child_process, // ECHILD
no_link,          // ENOLINK
no_lock_available, // ENOLCK
no_message_available, // ENODATA
no_message,       // ENOMSG
no_protocol_option, // ENOPROTOOPT
no_space_on_device, // ENOSPC
no_stream_resources, // ENOSR
no_such_device_or_address, // ENXIO
no_such_device,   // ENODEV
no_such_file_or_directory, // ENOENT
no_such_process,  // ESRCH
not_a_directory,  // ENOTDIR
not_a_socket,     // ENOTSOCK
not_a_stream,     // ENOSTR
not_connected,    // ENOTCONN
not_enough_memory, // ENOMEM
not_supported,    // ENOTSUP
operation_canceled, // EINPROGRESS
operation_not_permitted, // EPERM
operation_not_supported, // EWOULDBLOCK
owner_dead,       // EOWNERDEAD
permission_denied, // EPERM
protocol_error,   // EPROTYPE
protocol_not_supported, // EPROTONOSUPPORT
read_only_file_system, // EROFS
resource_deadlock_would_occur, // EDEADLK
resource_unavailable_try_again, // EAGAIN
result_out_of_range, // EROFS
state_not_recoverable, // ENOMSG
stream_timeout,   // ENOSPC
text_file_busy,   // ENOSPC
timed_out,        // ETIMEDOUT
too_many_files_open_in_system, // ENFILE
too_many_files_open, // EMFILE
too_many_links,   // EMLINK
too_many_symbolic_link_levels, // ELOOP
value_too_large,  // EOVERFLOW
wrong_protocol_type, // EPROTOTYPE
};

template<> struct is_error_condition_enum<errc> : true_type {};

// 19.5.4.5, non-member functions
error_code make_error_code(errc e) noexcept;

template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<<basic_ostream<charT, traits>& os, const error_code& ec);

// 19.5.5.5, non-member functions
error_condition make_error_condition(errc e) noexcept;
// 19.5.6, comparison operator functions
bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;
strong_ordering operator<=>(const error_code& lhs, const error_code& rhs) noexcept;
strong_ordering operator<=>(const error_condition& lhs, const error_condition& rhs) noexcept;

// 19.5.7, hash support
template<class T> struct hash;
template<> struct hash<error_code>;
template<> struct hash<error_condition>;

// 19.5, system error support
template<class T>
inline constexpr bool is_error_code_enum_v = is_error_code_enum<T>::value;
template<class T>
inline constexpr bool is_error_condition_enum_v = is_error_condition_enum<T>::value;

1 The value of each enum errc constant shall be the same as the value of the <cerrno> macro shown in the above synopsis. Whether or not the <system_error> implementation exposes the <cerrno> macros is unspecified.

2 The is_error_code_enum and is_error_condition_enum may be specialized for program-defined types to indicate that such types are eligible for class error_code and class error_condition automatic conversions, respectively.

19.5.3 Class error_category

19.5.3.1 Overview

The class error_category serves as a base class for types used to identify the source and encoding of a particular category of error code. Classes may be derived from error_category to support categories of errors in addition to those defined in this document. Such classes shall behave as specified in subclause 19.5.3.

[Note 1: error_category objects are passed by reference, and two such objects are equal if they have the same address. If there is more than a single object of a custom error_category type, such equality comparisons can evaluate to false even for objects holding the same value. — end note]
	namespace std {
    class error_category {
        public:
            constexpr error_category() noexcept;
            virtual ~error_category();
            error_category(const_error_category&) = delete;
            error_category& operator=(const error_category&) = delete;
            virtual const char* name() const noexcept = 0;
            virtual error_condition default_error_condition(int ev) const noexcept;
            virtual bool equivalent(int code, const error_condition& condition) const noexcept;
            virtual bool equivalent(const error_code& code, int condition) const noexcept;
            virtual string message(int ev) const = 0;

            bool operator==(const error_category& rhs) const noexcept;
            strong_ordering operator<=>(const error_category& rhs) const noexcept;
        }

        const error_category& generic_category() noexcept;
        const error_category& system_category() noexcept;
    }
}

19.5.3.2 Virtual members

virtual const char* name() const noexcept = 0;

1 Returns: A string naming the error category.

virtual error_condition default_error_condition(int ev) const noexcept;

2 Returns: error_condition(ev, *this).

§ 19.5.3.2
virtual bool equivalent(int code, const error_condition& condition) const noexcept;
3  
Returns: default_error_condition(code) == condition.

virtual bool equivalent(const error_code& code, int condition) const noexcept;
4  
Returns: *this == code.category() && code.value() == condition.

virtual string message(int ev) const = 0;
5  
Returns: A string that describes the error condition denoted by ev.

19.5.3.3 Non-virtual members

bool operator==(const error_category& rhs) const noexcept;
1  
Returns: this == &rhs.

strong_ordering operator<=>(const error_category& rhs) const noexcept;
2  
Returns: compare_three_way()(this, &rhs).

[Note 1: compare_three_way (20.14.8.8) provides a total ordering for pointers. — end note]

19.5.3.4 Program-defined classes derived from error_category

virtual const char* name() const noexcept = 0;
1  
Returns: A string naming the error category.

virtual error_condition default_error_condition(int ev) const noexcept;
2  
Returns: An object of type error_condition that corresponds to ev.

virtual bool equivalent(int code, const error_condition& condition) const noexcept;
3  
Returns: true if, for the category of error represented by *this, code is considered equivalent to condition; otherwise, false.

virtual bool equivalent(const error_code& code, int condition) const noexcept;
4  
Returns: true if, for the category of error represented by *this, code is considered equivalent to condition; otherwise, false.

19.5.3.5 Error category objects

const error_category& generic_category() noexcept;
1  
Returns: A reference to an object of a type derived from class error_category. All calls to this function shall return references to the same object.

Remarks: The object’s default_error_condition and equivalent virtual functions shall behave as specified for the class error_category. The object’s name virtual function shall return a pointer to the string "generic".

const error_category& system_category() noexcept;
3  
Returns: A reference to an object of a type derived from class error_category. All calls to this function shall return references to the same object.

Remarks: The object’s equivalent virtual functions shall behave as specified for class error_category. The object’s name virtual function shall return a pointer to the string "system". The object’s default_error_condition virtual function shall behave as follows:

If the argument ev corresponds to a POSIX errno value posv, the function shall return error_condition(posv, generic_category()). Otherwise, the function shall return error_condition(ev, system_category()). What constitutes correspondence for any given operating system is unspecified.

[Note 1: The number of potential system error codes is large and unbounded, and some might not correspond to any POSIX errno value. Thus implementations are given latitude in determining correspondence. — end note]

§ 19.5.3.5
19.5.4 Class error_code

19.5.4.1 Overview

The class error_code describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces.

[Note 1: Class error_code is an adjunct to error reporting by exception. — end note]

namespace std {
  class error_code {
    public:
      // 19.5.4.2, constructors
      error_code() noexcept;
      error_code(int val, const error_category& cat) noexcept;
      template<class ErrorCodeEnum>
        error_code(ErrorCodeEnum e) noexcept;

      // 19.5.4.3, modifiers
      void assign(int val, const error_category& cat) noexcept;
      template<class ErrorCodeEnum>
        error_code& operator=(ErrorCodeEnum e) noexcept;
      void clear() noexcept;

      // 19.5.4.4, observers
      int value() const noexcept;
      const error_category& category() const noexcept;
      error_condition default_error_condition() const noexcept;
      string message() const;
      explicit operator bool() const noexcept;
    private:
      int val_;       // exposition only
      const error_category* cat_; // exposition only
  };

  // 19.5.4.5, non-member functions
  error_code make_error_code(errc e) noexcept;

  template<class charT, class traits>
    basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& os, const error_code& ec);
}

19.5.4.2 Constructors

error_code() noexcept;
1
Postconditions: val_ == 0 and cat_ == &system_category().

error_code(int val, const error_category& cat) noexcept;
2
Postconditions: val_ == val and cat_ == &cat.

template<class ErrorCodeEnum>
  error_code(ErrorCodeEnum e) noexcept;
3
  Constraints: is_error_code_enum_v<ErrorCodeEnum> is true.
4
Postconditions: *this == make_error_code(e).

19.5.4.3 Modifiers

void assign(int val, const error_category& cat) noexcept;
1
  Postconditions: val_ == val and cat_ == &cat.

template<class ErrorCodeEnum>
  error_code& operator=(ErrorCodeEnum e) noexcept;
2
  Constraints: is_error_code_enum_v<ErrorCodeEnum> is true.

§ 19.5.4.3
 void clear() noexcept;

 Postconditions: value() == 0 and category() == system_category().

 19.5.4.4 Observers

 int value() const noexcept;

 Returns: val_.

 const error_category& category() const noexcept;

 Returns: *cat_.

 error_condition default_error_condition() const noexcept;

 Returns: category().default_error_condition(value()).

 string message() const;

 Returns: category().message(value()).

 explicit operator bool() const noexcept;

 Returns: value() != 0.

 19.5.4.5 Non-member functions

 error_code make_error_code(errc e) noexcept;

 Returns: error_code(static_cast<int>(e), generic_category()).

 template<class charT, class traits>
 basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, const error_code& ec);

 Effects: Equivalent to: return os << ec.category().name() << ':' << ec.value();

 19.5.5 Class error_condition

 19.5.5.1 Overview

 The class error_condition describes an object used to hold values identifying error conditions.

 [Note 1: error_condition values are portable abstractions, while error_code values (19.5.4) are implementation specific. —end note]

 namespace std {
  class error_condition {
   public:
    // 19.5.5.2, constructors
    error_condition() noexcept;
    error_condition(int val, const error_category& cat) noexcept;
    template<class ErrorConditionEnum>
      error_condition(ErrorConditionEnum e) noexcept;

    // 19.5.5.3, modifiers
    void assign(int val, const error_category& cat) noexcept;
    template<class ErrorConditionEnum>
      error_condition& operator=(ErrorConditionEnum e) noexcept;
    void clear() noexcept;

    // 19.5.5.4, observers
    int value() const noexcept;
    const error_category& category() const noexcept;
    string message() const;
    explicit operator bool() const noexcept;

  } // class error_condition

} // namespace std
private:
    int val_;    // exposition only
    const error_category* cat_; // exposition only
};

19.5.5.2 Constructors
error_condition() noexcept;
    Postconditions: val_ == 0 and cat_ == &generic_category().

error_condition(int val, const error_category& cat) noexcept;
    Postconditions: val_ == val and cat_ == &cat.

template<class ErrorConditionEnum>
    error_condition(ErrorConditionEnum e) noexcept;
    Constraints: is_error_condition_enum_v<ErrorConditionEnum> is true.
    Postconditions: this == make_error_condition(e).

19.5.5.3 Modifiers
void assign(int val, const error_category& cat) noexcept;
    Postconditions: val_ == val and cat_ == &cat.

template<class ErrorConditionEnum>
    error_condition& operator=(ErrorConditionEnum e) noexcept;
    Constraints: is_error_condition_enum_v<ErrorConditionEnum> is true.
    Postconditions: *this == make_error_condition(e).
    Returns: *this.

void clear() noexcept;
    Postconditions: value() == 0 and category() == generic_category().

19.5.5.4 Observers
int value() const noexcept;
    Returns: val_.

const error_category& category() const noexcept;
    Returns: *cat_.

string message() const;
    Returns: category().message(value()).

explicit operator bool() const noexcept;
    Returns: value() != 0.

19.5.5.5 Non-member functions
error_condition make_error_condition(errc e) noexcept;
    Returns: error_condition(static_cast<int>(e), generic_category()).

19.5.6 Comparison operator functions
bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
    Returns:
    lhs.category() == rhs.category() && lhs.value() == rhs.value()
bool operator==(const error_code& lhs, const error_condition& rhs) noexcept;

Returns:
  lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs, rhs.value())

bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;

Returns:
  lhs.category() == rhs.category() && lhs.value() == rhs.value()

strong_ordering operator<=>(const error_code& lhs, const error_code& rhs) noexcept;

Effects: Equivalent to:
  if (auto c = lhs.category() <=> rhs.category(); c != 0) return c;
  return lhs.value() <=> rhs.value();

strong_ordering operator<=>(const error_condition& lhs, const error_condition& rhs) noexcept;

Returns:
  if (auto c = lhs.category() <=> rhs.category(); c != 0) return c;
  return lhs.value() <=> rhs.value();

19.5.7 System error hash support

template<> struct hash<error_code>;
template<> struct hash<error_condition>;

1 The specializations are enabled (20.14.19).

19.5.8 Class system_error

19.5.8.1 Overview

The class system_error describes an exception object used to report error conditions that have an associated error code. Such error conditions typically originate from the operating system or other low-level application program interfaces.

2 [Note 1: If an error represents an out-of-memory condition, implementations are encouraged to throw an exception object of type bad_alloc (17.6.4.1) rather than system_error. — end note]

namespace std {
  class system_error : public runtime_error {
  public:
    system_error(error_code ec, const string& what_arg);
    system_error(error_code ec, const char* what_arg);
    system_error(error_code ec);
    system_error(int ev, const error_category& ecat, const string& what_arg);
    system_error(int ev, const error_category& ecat, const char* what_arg);
    system_error(int ev, const error_category& ecat);
    const error_code& code() const noexcept;
    const char* what() const noexcept override;
  }
};

19.5.8.2 Members

system_error(error_code ec, const string& what_arg);

1 Postconditions: code() == ec and
  string_view(what()).find(what_arg.c_str()) != string_view::npos.

system_error(error_code ec, const char* what_arg);

2 Postconditions: code() == ec and string_view(what()).find(what_arg) != string_view::npos.

system_error(error_code ec);

3 Postconditions: code() == ec.
system_error(int ev, const error_category& ecat, const string& what_arg);

Postconditions: code() == error_code(ev, ecat) and string_view(what()).find(what_arg.c_str()) != string_view::npos.

system_error(int ev, const error_category& ecat, const char* what_arg);

Postconditions: code() == error_code(ev, ecat) and string_view(what()).find(what_arg) != string_view::npos.

system_error(int ev, const error_category& ecat);

Postconditions: code() == error_code(ev, ecat).

const error_code& code() const noexcept;

Returns: ec or error_code(ev, ecat), from the constructor, as appropriate.

const char* what() const noexcept override;

Returns: An ntbs incorporating the arguments supplied in the constructor.

[Note 1: The returned ntbs might be the contents of what_arg + " : " + code.message(). — end note]
20 General utilities library

20.1 General

This Clause describes utilities that are generally useful in C++ programs; some of these utilities are used by other elements of the C++ standard library. These utilities are summarized in Table 41.

Table 41: General utilities library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.2 Utility components</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>20.3 Compile-time integer sequences</td>
<td></td>
</tr>
<tr>
<td>20.4 Pairs</td>
<td></td>
</tr>
<tr>
<td>20.5 Tuples</td>
<td>&lt;tuple&gt;</td>
</tr>
<tr>
<td>20.6 Optional objects</td>
<td>&lt;optional&gt;</td>
</tr>
<tr>
<td>20.7 Variants</td>
<td>&lt;variant&gt;</td>
</tr>
<tr>
<td>20.8 Storage for any type</td>
<td>&lt;any&gt;</td>
</tr>
<tr>
<td>20.9 Fixed-size sequences of bits</td>
<td>&lt;bitset&gt;</td>
</tr>
<tr>
<td>20.10 Memory</td>
<td>&lt;cstdlib&gt;, &lt;memory&gt;</td>
</tr>
<tr>
<td>20.11 Smart pointers</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>20.12 Memory resources</td>
<td>&lt;memory_resource&gt;</td>
</tr>
<tr>
<td>20.13 Scoped allocators</td>
<td>&lt;scoped_allocator&gt;</td>
</tr>
<tr>
<td>20.14 Function objects</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>20.15 Type traits</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>20.16 Compile-time rational arithmetic</td>
<td>&lt;ratio&gt;</td>
</tr>
<tr>
<td>20.17 Type indexes</td>
<td>&lt;typeindex&gt;</td>
</tr>
<tr>
<td>20.18 Execution policies</td>
<td>&lt;execution&gt;</td>
</tr>
<tr>
<td>20.19 Primitive numeric conversions</td>
<td>&lt;charconv&gt;</td>
</tr>
<tr>
<td>20.20 Formatting</td>
<td>&lt;format&gt;</td>
</tr>
</tbody>
</table>

20.2 Utility components

20.2.1 Header <utility> synopsis

The header <utility> contains some basic function and class templates that are used throughout the rest of the library.

```cpp
#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.2

namespace std {

// 20.2.2, swap
  template<class T>
    constexpr void swap(T& a, T& b) noexcept(see below);
  template<class T, size_t N>
    constexpr void swap(T (&a)[N], T (&b)[N]) noexcept(is_nothrow_swappable_v<T>);

// 20.2.3, exchange
  template<class T, class U = T>
    constexpr T exchange(T& obj, U&& new_val);

// 20.2.4, forward/move
  template<class T>
    constexpr T& forward(remove_reference_t<T>& t) noexcept;
  template<class T>
    constexpr T& forward(remove_reference_t<T>&& t) noexcept;
  template<class T>
    constexpr remove_reference_t<T>&& move(T&&) noexcept;
```

§ 20.2.1
template<class T>
constexpr conditional_t<!is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&, T&&>
move_if_noexcept(T& x) noexcept;

// 20.2.5, as_const
template<class T>
constexpr add_const_t<T>& as_const(T& t) noexcept;
template<class T>
void as_const(const T&&) = delete;

// 20.2.6, declval
template<class T>
add_rvalue_reference_t<T> declval() noexcept; // as unevaluated operand

// 20.2.7, integer comparison functions
template<class T, class U>
constexpr bool cmp_equal(T t, U u) noexcept;
template<class T, class U>
constexpr bool cmp_not_equal(T t, U u) noexcept;
template<class T, class U>
constexpr bool cmp_less(T t, U u) noexcept;
template<class T, class U>
constexpr bool cmp_greater(T t, U u) noexcept;
template<class T, class U>
constexpr bool cmp_less_equal(T t, U u) noexcept;
template<class T, class U>
constexpr bool cmp_greater_equal(T t, U u) noexcept;

// 20.3, compile-time integer sequences
template<class T, T...>
struct integer_sequence;
template<size_t... I>
using index_sequence = integer_sequence<size_t, I...>;
template<class T, T N>
using make_integer_sequence = integer_sequence<T, see below>;
template<size_t N>
using make_index_sequence = make_integer_sequence<size_t, N>;
template<class... T>
using index_sequence_for = make_index_sequence<sizeof...(T)>;

// 20.4, class template pair
template<class T1, class T2>
struct pair;

// 20.4.3, pair specialized algorithms
template<class T1, class T2>
constexpr bool operator==(const pair<T1, T2>&, const pair<T1, T2>&);
template<class T1, class T2>
constexpr common_comparison_category_t<
    synth-three-way-result<T1>,
    synth-three-way-result<T2>>
operator<=>(const pair<T1, T2>&, const pair<T1, T2>&);

template<class T1, class T2>
constexpr void swap(pair<T1, T2>& x, pair<T1, T2>& y) noexcept(noexcept(x.swap(y)));

template<class T1, class T2>
constexpr see below make_pair(T1&&, T2&&);
// 20.4.4, tuple-like access to pair
template<class T> struct tuple_size;
template<
size_t I, class T> struct tuple_element;

template<class T1, class T2> struct tuple_size<pair<T1, T2>>;
template<
size_t I, class T1, class T2> struct tuple_element<pair<T1, T2>, I, I>;

template<
size_t I, class T1, class T2> constexpr tuple_element_t<I, pair<T1, T2>>& get(pair<T1, T2>&) noexcept;
template<
size_t I, class T1, class T2> constexpr tuple_element_t<I, pair<T1, T2>>& get(pair<T1, T2>&&) noexcept;
template<
size_t I, class T1, class T2> constexpr const tuple_element_t<I, pair<T1, T2>>& get(const pair<T1, T2>&) noexcept;
template<
size_t I, class T1, class T2> constexpr const tuple_element_t<I, pair<T1, T2>>& get(const pair<T1, T2>&&) noexcept;

template<class T1, class T2> constexpr T1& get(pair<T1, T2>& p) noexcept;
template<class T1, class T2> constexpr const T1& get(const pair<T1, T2>& p) noexcept;
template<class T1, class T2> constexpr T1&& get(pair<T1, T2>&& p) noexcept;
template<class T1, class T2> constexpr const T1&& get(const pair<T1, T2>&& p) noexcept;

template<class T2, class T1> constexpr T2& get(pair<T1, T2>& p) noexcept;
template<class T2, class T1> constexpr const T2& get(const pair<T1, T2>& p) noexcept;
template<class T2, class T1> constexpr T2&& get(pair<T1, T2>&& p) noexcept;
template<class T2, class T1> constexpr const T2&& get(const pair<T1, T2>&& p) noexcept;

// 20.4.5, pair piecewise construction
struct piecewise_construct_t {
    explicit piecewise_construct_t() = default;
};
inline constexpr piecewise_construct_t piecewise_construct{};

template<class... Types> class tuple; // defined in <tuple> (20.5.2)

// in-place construction
struct in_place_t {
    explicit in_place_t() = default;
};
inline constexpr in_place_t in_place{};

template<class T>
struct in_place_type_t {
    explicit in_place_type_t() = default;
};
inline constexpr in_place_type_t<T> in_place_type{};

template<
size_t I> struct in_place_index_t {
    explicit in_place_index_t() = default;
};
inline constexpr in_place_index_t<I> in_place_index{};

20.2.2 swap

template<class T>
constexpr void swap(T& a, T& b) noexcept(see below);

Constraints: is_move_constructible_v<T> is true and is_move_assignable_v<T> is true.
Preconditions: Type T meets the Cpp17MoveConstructible (Table 28) and Cpp17MoveAssignable (Table 30) requirements.

Effects: Exchanges values stored in two locations.

Remarks: This function is a designated customization point (16.4.5.2.1). The expression inside noexcept is equivalent to:

\[
is_{\text{nothrow\_move\_constructible\_v}}(T) \&\& is_{\text{nothrow\_move\_assignable\_v}}(T)
\]

```cpp
template<class T, size_t N>
constexpr void swap(T (&a)[N], T (&b)[N]) noexcept(is_nothrow_swappable_v<T>);
```

Constraints: is_swappable_v<T> is true.

Preconditions: a[i] is swappable with (16.4.4.3) b[i] for all i in the range [0, N).

Effects: As if by swap_ranges(a, a + N, b).

20.2.3 exchange

```cpp
template<class T, class U = T>
constexpr T exchange(T& obj, U&& new_val);
```

Effects: Equivalent to:

\[
T old_val = std::move(obj);
obj = std::forward<U>(new_val);
return old_val;
\]

20.2.4 Forward/move helpers

The library provides templated helper functions to simplify applying move semantics to an lvalue and to simplify the implementation of forwarding functions. All functions specified in this subclause are signal-safe (17.13.5).

```cpp
template<class T> constexpr T&& forward(remove_reference_t<T>& t) noexcept;
template<class T> constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
```

Mandates: For the second overload, is_lvalue_reference_v<T> is false.

Returns: static_cast<T&&>(t).

[Example 1:]

```cpp
template<class T, class A1, class A2>
shared_ptr<T> factory(A1&& a1, A2&& a2) {
    return shared_ptr<T>(new T(std::forward<A1>(a1), std::forward<A2>(a2)));
}
```

```cpp
struct A {
    A(int&, const double&);
};

void g() {
    shared_ptr<A> sp1 = factory<A>(2, 1.414); // error: 2 will not bind to int&
    int i = 2;
    shared_ptr<A> sp2 = factory<A>(i, 1.414); // OK
}
```

In the first call to `factory`, A1 is deduced as int, so 2 is forwarded to A's constructor as an rvalue. In the second call to `factory`, A1 is deduced as int&, so i is forwarded to A's constructor as an lvalue. In both cases, A2 is deduced as double, so 1.414 is forwarded to A's constructor as an rvalue. — end example

```cpp
template<class T> constexpr remove_reference_t<T>&& move(T&& t) noexcept;
```

Returns: static_cast<remove_reference_t<T>&&>(t).

[Example 2:]

```cpp
template<class T, class A1>
shared_ptr<T> factory(A1&& a1) {
    return shared_ptr<T>(new T(std::forward<A1>(a1)));
}
```
struct A {
    A();  // copies from lvalues
    A(const A&);  // moves from rvalues
    A(A&&);  // moves from rvalues
};

void g() {
    A a;
    shared_ptr<A> sp1 = factory<A>(a);  // "a" binds to A(const A&)
    shared_ptr<A> sp1 = factory<A>(std::move(a));  // "a" binds to A(A&&)
}

In the first call to factory, A1 is deduced as A&, so a is forwarded as a non-const lvalue. This binds to the constructor A(const A&), which copies the value from a. In the second call to factory, because of the call std::move(a), A1 is deduced as A, so a is forwarded as an rvalue. This binds to the constructor A(A&&), which moves the value from a. —end example

template<class T> constexpr conditional_t<!is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&, T&&> move_if_noexcept(T& x) noexcept;

20.2.5 Function template as_const

template<class T> constexpr add_const_t<T>& as_const(T& t) noexcept;

20.2.6 Function template declval

template<class T> constexpr add_rvalue_reference_t<T> declval() noexcept;  // as unevaluated operand

Mandates: This function is not odr-used (6.3).
Remarks: The template parameter T of declval may be an incomplete type.

[Example 1: template<class To, class From> decltype(static_cast<To>(declval<From>())) convert(From&&);
 declares a function template convert which only participates in overload resolution if the type From can be explicitly converted to type To. For another example see class template common_type (20.15.8.7). —end example]

20.2.7 Integer comparison functions

template<class T, class U>
constexpr bool cmp_equal(T t, U u) noexcept;

Mandates: Both T and U are standard integer types or extended integer types (6.8.2).

Effects: Equivalent to:

using UT = make_unsigned_t<T>;
using UU = make_unsigned_t<U>;
if constexpr (is_signed_v<T> == is_signed_v<U>)
    return t == u;
else if constexpr (is_signed_v<T>)
    return t < 0 ? false : UT(t) == u;
else
    return u < 0 ? false : t == UU(u);

template<class T, class U>
constexpr bool cmp_not_equal(T t, U u) noexcept;

Effects: Equivalent to: return !cmp_equal(t, u);
template<class T, class U>
    constexpr bool cmp_less(T t, U u) noexcept;

Mandates: Both T and U are standard integer types or extended integer types (6.8.2).

Effects: Equivalent to:

    using UT = make_unsigned_t<T>;
    using UU = make_unsigned_t<U>;
    if constexpr (is_signed_v<T> == is_signed_v<U>)
        return t < u;
    else if constexpr (is_signed_v<T>)
        return t < 0 ? true : UT(t) < u;
    else
        return u < 0 ? false : t < UU(u);

template<class T, class U>
    constexpr bool cmp_greater(T t, U u) noexcept;

Effects: Equivalent to: return cmp_less(u, t);

template<class T, class U>
    constexpr bool cmp_less_equal(T t, U u) noexcept;

Effects: Equivalent to:

    return !cmp_greater(t, u);

template<class T, class U>
    constexpr bool cmp_greater_equal(T t, U u) noexcept;

Effects: Equivalent to:

    return !cmp_less(t, u);

template<class R, class T>
    constexpr bool in_range(T t) noexcept;

Mandates: Both T and R are standard integer types or extended integer types (6.8.2).

Effects: Equivalent to:

    return cmp_greater_equal(t, numeric_limits<R>::min()) &&
          cmp_less_equal(t, numeric_limits<R>::max());

[Note 1: These function templates cannot be used to compare byte, char, char8_t, char16_t, char32_t, wchar_t, and bool. —end note]

20.3 Compile-time integer sequences [intseq]

20.3.1 In general [intseq.general]

The library provides a class template that can represent an integer sequence. When used as an argument to a function template the template parameter pack defining the sequence can be deduced and used in a pack expansion.

[Note 1: The index_sequence alias template is provided for the common case of an integer sequence of type size_t; see also 20.5.5. —end note]

20.3.2 Class template integer_sequence [intseq.intseq]

namespace std {
    template<class T, T... I> struct integer_sequence {
        using value_type = T;
        static constexpr size_t size() noexcept { return sizeof...(I); }
    };
}

Mandates: T is an integer type.

20.3.3 Alias template make_integer_sequence [intseq.make]

template<class T, T N>
    using make_integer_sequence = integer_sequence<T, see below>;

Mandates: N ≥ 0.
The alias template `make_integer_sequence` denotes a specialization of `integer_sequence` with \( N \) non-type template arguments. The type `make_integer_sequence\langle T, N \rangle` is an alias for the type `integer_sequence\langle T, 0, 1, \ldots, N-1 \rangle`.

[Note 1: `make_integer_sequence\langle int, 0 \rangle` is an alias for the type `integer_sequence\langle int \rangle`. — end note]

### 20.4 Pairs

#### 20.4.1 In general

The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to `pair` objects as if they were `tuple` objects (see 20.5.6 and 20.5.7).

#### 20.4.2 Class template pair

```cpp
namespace std {
    template<class T1, class T2>
    struct pair {
        using first_type = T1;
        using second_type = T2;

        T1 first;
        T2 second;

        pair(const pair&) = default;
        pair(pair&&) = default;
        constexpr explicit(pair(const T1& x, const T2& y));
        template<class U1, class U2>
        constexpr explicit(pair(U1&& x, U2&& y));
        template<class U1, class U2>
        constexpr explicit(pair(const pair<U1, U2>& p));
        template<class U1, class U2>
        constexpr explicit(pair(pair<U1, U2>&& p));
        template<class... Args1, class... Args2>
        constexpr pair(piecewise_construct_t, tuple<Args1...> first_args, tuple<Args2...> second_args);

        constexpr pair& operator=(const pair& p);
        template<class U1, class U2>
        constexpr pair& operator=(const pair<U1, U2>& p);
        constexpr pair& operator=(pair&& p) noexcept;
        template<class U1, class U2>
        constexpr pair& operator=(pair<U1, U2>&& p);

        constexpr void swap(pair& p) noexcept;
    };

    template<class T1, class T2>
    pair(T1, T2) -> pair<T1, T2>;
}
```

1 Constructors and member functions of `pair` do not throw exceptions unless one of the element-wise operations specified to be called for that operation throws an exception.

2 The defaulted move and copy constructor, respectively, of `pair` is a `constexpr` function if and only if all required element-wise initializations for move and copy, respectively, would satisfy the requirements for a `constexpr` function.

3 If `(is_trivially_destructible_v<T1> && is_trivially_destructible_v<T2>)` is `true`, then the destructor of `pair` is trivial.

4 `pair<T, U>` is a structural type (13.2) if `T` and `U` are both structural types. Two values `p1` and `p2` of type `pair<T, U>` are template-argument-equivalent (13.6) if and only if `p1.first` and `p2.first` are template-argument-equivalent and `p1.second` and `p2.second` are template-argument-equivalent.
constexpr explicit(see below) pair();

Constraints:

(5.1)  
— is_default_constructible_v<first_type> is true and

(5.2)  
— is_default_constructible_v<second_type> is true.

Effects: Value-initializes first and second.

Remarks: The expression inside explicit evaluates to true if and only if either first_type or second_type is not implicitly default-constructible.

[Note 1: This behavior can be implemented with a trait that checks whether a const first_type& or a const second_type& can be initialized with {}.
—end note]

constexpr explicit(see below) pair(const T1& x, const T2& y);

Constraints:

(8.1)  
— is_copy_constructible_v<first_type> is true and

(8.2)  
— is_copy_constructible_v<second_type> is true.

Effects: Initializes first with x and second with y.

Remarks: The expression inside explicit is equivalent to:

!is_convertible_v<const first_type&, first_type> ||
!is_convertible_v<const second_type&, second_type>

template<class U1, class U2> constexpr explicit(see below) pair(U1&& x, U2&& y);

Constraints:

(11.1)  
— is_constructible_v<first_type, U1> is true and

(11.2)  
— is_constructible_v<second_type, U2> is true.

Effects: Initializes members from the corresponding members of the argument.

Remarks: The expression inside explicit is equivalent to:

!is_convertible_v<U1, first_type> || !is_convertible_v<U2, second_type>

template<class U1, class U2> constexpr explicit(see below) pair(const pair<U1, U2>& p);

Constraints:

(14.1)  
— is_constructible_v<first_type, const U1&> is true and

(14.2)  
— is_constructible_v<second_type, const U2&> is true.

Effects: Initializes members from the corresponding members of the argument.

Remarks: The expression inside explicit is equivalent to:

!is_convertible_v<const U1&, first_type> || !is_convertible_v<const U2&, second_type>

template<class U1, class U2> constexpr explicit(see below) pair(pair<U1, U2>&& p);

Constraints:

(17.1)  
— is_constructible_v<first_type, U1> is true and

(17.2)  
— is_constructible_v<second_type, U2> is true.

Effects: Initializes first with std::forward<U1>(p.first) and second with std::forward<U2>(p.second).

Remarks: The expression inside explicit is equivalent to:

!is_convertible_v<U1, first_type> || !is_convertible_v<U2, second_type>

template<class... Args1, class... Args2>
constexpr pair(piecewise_construct_t,
        tuple<Args1...> first_args, tuple<Args2...> second_args);

Mandates:

(20.1)  
— is_constructible_v<first_type, Args1...> is true and

§ 20.4.2
20.4.3 Specialized algorithms [pairs.spec]

```cpp
template<class T1, class T2>
constexpr bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

Returns: `x.first == y.first && x.second == y.second`.

---

21

Effects: Initializes `first` with arguments of types `Args1...` obtained by forwarding the elements of `first_args` and initializes `second` with arguments of types `Args2...` obtained by forwarding the elements of `second_args`. (Here, forwarding an element `x` of type `U` within a `tuple` object means calling `std::forward<U>(x)`.) This form of construction, whereby constructor arguments for `first` and `second` are each provided in a separate `tuple` object, is called **piecewise construction**.

```cpp
constexpr pair& operator=(const pair& p);
```

Effects: Assigns `p.first` to `first` and `p.second` to `second`.

Returns: `*this`.

Remarks: This operator is defined as deleted unless `is_copyAssignable_v<first_type>` is true and `is_copyAssignable_v<second_type>` is true.

```cpp
template<class U1, class U2> constexpr pair& operator=(const pair<U1, U2>& p);
```

Effects: Assigns `p.first` to `first` and `p.second` to `second`.

Returns: `*this`.

Constraints:
- `(25.1) is_assignable_v<first_type&, const U1&>` is true and
- `(25.2) is_assignable_v<second_type&, const U2&>` is true.

Remarks: The expression inside `noexcept` is equivalent to:

```cpp
is_nothrow_moveAssignable_v<T1> && is_nothrow_moveAssignable_v<T2>
```

```cpp
template<class U1, class U2> constexpr pair& operator=(pair<U1, U2>&& p);
```

Effects: Assigns to `first` with `std::forward<U1>(p.first)` and to `second` with `std::forward<U2>(p.second)`.

Returns: `*this`.

Constraints:
- `(32.1) is_assignable_v<first_type&, U1>` is true and
- `(32.2) is_assignable_v<second_type&, U2>` is true.

Remarks: The expression inside `noexcept` is equivalent to:

```cpp
is_nothrow_moveAssignable_v<first_type> && is_nothrow_moveAssignable_v<second_type>
```

```cpp
constexpr void swap(pair& p) noexcept(see below);
```

Preconditions: `first` is swappable with (16.4.4.3) `p.first` and `second` is swappable with `p.second`.

Effects: Swaps `first` with `p.first` and `second` with `p.second`.

Remarks: The expression inside `noexcept` is equivalent to:

```cpp
is_nothrow_swapAssignable_v<first_type> && is_nothrow_swapAssignable_v<second_type>
```
template<class T1, class T2>
constexpr common_comparison_category_t<
synth-three-way-result<T1>,
synth-three-way-result<T2>>
operator<=>(const pair<T1, T2>& x, const pair<T1, T2>& y);

Effects: Equivalent to:
   if (auto c = synth-three-way(x.first, y.first); c != 0) return c;
   return synth-three-way(x.second, y.second);

template<class T1, class T2>
constexpr void swap(pair<T1, T2>& x, pair<T1, T2>& y) noexcept(noexcept(x.swap(y)));

Constraints: is_swappable_v<T1> is true and is_swappable_v<T2> is true.

Effects: Equivalent to x.swap(y).

template<class T1, class T2>
constexpr pair<unwrap_ref_decay_t<T1>, unwrap_ref_decay_t<T2>> make_pair(T1&& x, T2&& y);

Returns:
   pair<unwrap_ref_decay_t<T1>,
       unwrap_ref_decay_t<T2>>(std::forward<T1>(x), std::forward<T2>(y))

[Example 1: In place of:
   return pair<int, double>(5, 3.1415926); // explicit types
a C++ program may contain:
   return make_pair(5, 3.1415926); // types are deduced
   —end example]
template<class T1, class T2>
constexpr const T1&& get(const pair<T1, T2>&& p) noexcept;

Mandates: T1 and T2 are distinct types.

Returns: A reference to p.first.

template<class T2, class T1>
constexpr T2& get(pair<T1, T2>& p) noexcept;
template<class T2, class T1>
constexpr const T2& get(const pair<T1, T2>& p) noexcept;
template<class T2, class T1>
constexpr T2&& get(pair<T1, T2>&& p) noexcept;
template<class T2, class T1>
constexpr const T2&& get(const pair<T1, T2>&& p) noexcept;

Mandates: T1 and T2 are distinct types.

Returns: A reference to p.second.

### 20.4.5 Piecewise construction [pair.piecewise]

struct piecewise_construct_t {
    explicit piecewise_construct_t() = default;
};
inline constexpr piecewise_construct_t piecewise_construct{};

1 The struct piecewise_construct_t is an empty class type used as a unique type to disambiguate constructor and function overloading. Specifically, pair has a constructor with piecewise_construct_t as the first argument, immediately followed by two tuple (20.5) arguments used for piecewise construction of the elements of the pair object.

### 20.5 Tuples [tuple]

#### 20.5.1 In general [tuple.general]

1 Subclause 20.5 describes the tuple library that provides a tuple type as the class template tuple that can be instantiated with any number of arguments. Each template argument specifies the type of an element in the tuple. Consequently, tuples are heterogeneous, fixed-size collections of values. An instantiation of tuple with two arguments is similar to an instantiation of pair with the same two arguments. See 20.4.

#### 20.5.2 Header <tuple> synopsis [tuple.syn]

```c++
#include <compare> // see 17.11.1

namespace std {
    // 20.5.3, class template tuple
    template<class... Types>
    class tuple;

    // 20.5.4, tuple creation functions
    inline constexpr unspecified ignore;
    template<class... TTypes>
    constexpr tuple<unwrap_ref_decay_t<TTypes>...> make_tuple(TTypes&&...);

    template<class... TTypes>
    constexpr tuple<TTypes&&...> forward_as_tuple(TTypes&&...) noexcept;

    template<class... TTypes>
    constexpr tuple<TTypes&...> tie(TTypes&...) noexcept;

    template<class... Tuples>
    constexpr tuple<CTypes...> tuple_cat(Tuples&&...);

    // 20.5.5, calling a function with a tuple of arguments
    template<class F, class Tuple>
    constexpr decltype(auto) apply(F&& f, Tuple&& t);
}
```

§ 20.5.2 590
template<class T, class Tuple>
constexpr T make_from_tuple(Tuple&& t);

// 20.5.6, tuple helper classes

template<class T>
struct tuple_size;
// not defined

template<class... Types>
struct tuple_size<tuple<Types...>>;

template<size_t I, class T>
struct tuple_element;
// not defined

template<size_t I, class... Types>
struct tuple_element<I, tuple<Types...>>;

template<size_t I, class T>
using tuple_element_t = typename tuple_element<I, T>::type;

// 20.5.7, element access

template<size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>& get(tuple<Types...>&) noexcept;

template<size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>& get(tuple<Types...>&&) noexcept;

template<size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>>& get(const tuple<Types...>&) noexcept;

template<size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>>& get(const tuple<Types...>&&) noexcept;

// 20.5.8, relational operators

template<class... TTypes, class... UTypes>
constexpr bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);

// 20.5.9, allocator-related traits

template<class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc>;

// 20.5.10, specialized algorithms

template<class... Types>
constexpr void swap(tuple<Types...>& x, tuple<Types...>&& y) noexcept(see below);

// 20.5.6, tuple helper classes

template<class T>
inline constexpr size_t tuple_size_v = tuple_size<T>::value;

20.5.3 Class template tuple

namespace std {

template<class... Types>
class tuple {
public:

    // 20.5.3.1, tuple construction
    constexpr explicit(see below) tuple();

    // 20.5.6, tuple helper classes
    template<class T, class Tuple>
    constexpr T make_from_tuple(Tuple&& t);

    // 20.5.7, element access
    
    // 20.5.8, relational operators
    
    // 20.5.9, allocator-related traits
    
    // 20.5.10, specialized algorithms
}

§ 20.5.3
constexpr explicit(see below) tuple(const Types&...); // only if sizeof...(Types) >= 1

template<typename... UTypes>
constexpr explicit(see below) tuple(UTypes&&...); // only if sizeof...(Types) >= 1

tuple(const tuple&) = default;
tuple(tuple&&) = default;

template<typename... UTypes>
constexpr explicit(see below) tuple(const tuple<UTypes...>&);
template<typename... UTypes>
constexpr explicit(see below) tuple(const tuple<UTypes...>&&);

template<typename U1, typename U2>
constexpr explicit(see below) tuple(const pair<U1, U2>&); // only if sizeof...(Types) == 2

template<typename U1, typename U2>
constexpr explicit(see below) tuple(pair<U1, U2>&&); // only if sizeof...(Types) == 2

// allocator-extended constructors

template<typename Alloc>
constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a);

template<typename Alloc>
constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a, const Types&...);

template<typename Alloc, typename... UTypes>
constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a, UTypes&&...);

template<typename Alloc>
constexpr tuple(allocator_arg_t, const Alloc& a, const tuple&);

template<typename Alloc>
constexpr tuple(allocator_arg_t, const Alloc& a, tuple&&);

template<typename Alloc, typename... UTypes>
constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);

template<typename Alloc, typename... UTypes>
constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);

template<typename U1, typename U2>
constexpr explicit(see below) tuple(const pair<U1, U2>&); // only if sizeof...(Types) == 2

template<typename U1, typename U2>
constexpr explicit(see below) tuple(pair<U1, U2>&&); // only if sizeof...(Types) == 2

// 20.5.3.2. tuple assignment

constexpr tuple& operator=(const tuple&);
constexpr tuple& operator=(tuple&&) noexcept(see below);

template<typename... UTypes>
constexpr tuple& operator=(const tuple<UTypes...>&);

template<typename... UTypes>
constexpr tuple& operator=(tuple<UTypes...>&&);

template<typename U1, typename U2>
constexpr tuple& operator=(const pair<U1, U2>&); // only if sizeof...(Types) == 2

template<typename U1, typename U2>
constexpr tuple& operator=(pair<U1, U2>&&); // only if sizeof...(Types) == 2

// 20.5.3.3. tuple swap

constexpr void swap(tuple&) noexcept(see below);

};

template<typename... UTypes>
tuple<UTypes...> -> tuple<UTypes...>;
```cpp
    template<class T1, class T2>
    tuple(pair<T1, T2>) -> tuple<T1, T2>;
    template<class Alloc, class... UTypes>
    tuple(allocation_arg_t, Alloc, UTypes...) -> tuple<UTypes...>;
    template<class Alloc, class T1, class T2>
    tuple(allocation_arg_t, Alloc, pair<T1, T2>) -> tuple<T1, T2>;
    template<class Alloc, class... UTypes>
    tuple(allocation_arg_t, Alloc, tuple<UTypes...>) -> tuple<UTypes...>;
```

### 20.5.3.1 Construction

1. In the descriptions that follow, let \( i \) be in the range \([0, \text{sizeof}(\text{Types}))\) in order, \( T_i \) be the \( i \)th type in \( \text{Types} \), and \( U_i \) be the \( i \)th type in a template parameter pack named \( \text{UTypes} \), where indexing is zero-based.

2. For each `tuple` constructor, an exception is thrown only if the construction of one of the types in \( \text{Types} \) throws an exception.

3. The defaulted move and copy constructor, respectively, of `tuple` is a constexpr function if and only if all required element-wise initializations for move and copy, respectively, would satisfy the requirements for a constexpr function. The defaulted move and copy constructor of `tuple<>` are constexpr functions.

4. If `is_trivially_destructible_v<T_i>` is true for all \( T_i \), then the destructor of `tuple` is trivial.

```cpp
    constexpr explicit(\text{see below}) tuple();
```

5. **Constraints:** `is_default_constructible_v<T_i>` is true for all \( i \).

6. **Effects:** Value-initializes each element.

7. **Remarks:** The expression inside `explicit` evaluates to `true` if and only if \( T_i \) is not copy-list-initializable from an empty list for at least one \( i \).

   [Note 1: This behavior can be implemented with a trait that checks whether a `const T_i&` can be initialized with `{}`. —end note]

```cpp
    constexpr explicit(\text{see below}) tuple(const Types&...);
```

8. **Constraints:** `sizeof...(\text{Types}) \geq 1` and `is_copy_constructible_v<T_i>` is true for all \( i \).

9. **Effects:** Initializes each element with the value of the corresponding parameter.

10. **Remarks:** The expression inside `explicit` is equivalent to:

    ```cpp
    !\text{conjunction}_v\text{is_convertible<const Types&, Types>...} \text{tuple<>(}\text{see below}) \text{constexpr explicit}(\text{see below}) \text{tuple(const Types&...)};
    ```

```cpp
    template<class... UTypes> constexpr explicit(\text{see below}) \text{tuple<UTypes&&... u>};
```

11. **Constraints:** `sizeof...(\text{Types})` equals `sizeof...(\text{UTypes})` and `sizeof...(\text{Types}) \geq 1` and `is_constructible_v<T_i, const U_i&>` is true for all \( i \).

12. **Effects:** Initializes the elements in the tuple with the corresponding value in `std::forward<UTypes>(u)`.

13. **Remarks:** The expression inside `explicit` is equivalent to:

    ```cpp
    !\text{conjunction}_v\text{is_convertible<UTypes, Types>...} \text{tuple(const tuple<UTypes...>& u)};
    ```

```cpp
    tuple(const tuple& u) = default;
```

14. **Mandates:** `is_copy_constructible_v<T_i>` is true for all \( i \).

15. **Effects:** Initializes each element of `*this` with the corresponding element of \( u \).

```cpp
    tuple(tuple&& u) = default;
```

16. **Constraints:** `is_move_constructible_v<T_i>` is true for all \( i \).

17. **Effects:** For all \( i \), initializes the \( i \)th element of `*this` with `std::forward<T_i>(\text{get}<i>(u))`.

```cpp
    template<class... UTypes> constexpr explicit(\text{see below}) tuple(const tuple<UTypes...>& u);
```

18. **Constraints:**

   (18.1) `sizeof...(\text{Types})` equals `sizeof...(\text{UTypes})`

   (18.2) `is_constructible_v<T_i, const U_i&>` is true for all \( i \), and
— either sizeof...(Types) is not 1, or (when Types... expands to T and UTypes... expands to U) is_convertible_v<const tuple<U&>, T>, is_constructible_v<T, const tuple<U&>>, and is_same_v<T, U> are all false.

**Effects:** Initializes each element of *this with the corresponding element of u.

**Remarks:** The expression inside explicit is equivalent to:

```cpp
!conjunction_v<is_convertible<const UTypes&, Types>...>
```

```cpp
template<class... UTypes> constexpr explicit(see below) tuple(tuple<UTypes...>&& u);
```

**Constraints:**

1. sizeof...(Types) equals sizeof...(UTypes), and
2. is_constructible_v<T_i, U_i> is true for all i, and
3. either sizeof...(Types) is not 1, or (when Types... expands to T and UTypes... expands to U) is_convertible_v<tuple<U>, T>, is_constructible_v<T, tuple<U>>, and is_same_v<T, U> are all false.

**Effects:** For all i, initializes the i-th element of *this with std::forward<U_i>(get<i>(u)).

**Remarks:** The expression inside explicit is equivalent to:

```cpp
!conjunction_v<is_convertible<UTypes, Types>...>
```

```cpp
template<class U1, class U2> constexpr explicit(see below) tuple(const pair<U1, U2>& u);
```

**Constraints:**

1. sizeof...(Types) is 2,
2. is_constructible_v<T_0, const U1&> is true, and
3. is_constructible_v<T_1, const U2&> is true.

**Effects:** Initializes the first element with u.first and the second element with u.second.

**Remarks:** The expression inside explicit is equivalent to:

```cpp
!is_convertible_v<const U1&, T_0> || !is_convertible_v<const U2&, T_1>
```

```cpp
template<class U1, class U2> constexpr explicit(see below) tuple(pair<U1, U2>&& u);
```

**Constraints:**

1. sizeof...(Types) is 2,
2. is_constructible_v<T_0, U1> is true, and
3. is_constructible_v<T_1, U2> is true.

**Effects:** Initializes the first element with std::forward<U1>(u.first) and the second element with std::forward<U2>(u.second).

**Remarks:** The expression inside explicit is equivalent to:

```cpp
!is_convertible_v<U1, T_0> || !is_convertible_v<U2, T_1>
```

```cpp
template<class Alloc> constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a);
```

```cpp
template<class Alloc> constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, const Types&...);
```

```cpp
template<class Alloc, class... UTypes> constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
```

```cpp
template<class Alloc> constexpr tuple(allocator_arg_t, const Alloc& a, const tuple&);
```

```cpp
template<class Alloc> constexpr tuple(allocator_arg_t, const Alloc& a, tuple&&);
```

```cpp
template<class Alloc, class... UTypes> constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
```
template<class Alloc, class... UTypes>
constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);

template<class Alloc, class U1, class U2>
constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);

template<class Alloc, class U1, class U2>
constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

Preconditions: Alloc meets the Cpp17Allocator requirements (Table 36).

Effects: Equivalent to the preceding constructors except that each element is constructed with use-
allocator construction (20.10.8.2).

20.5.3.2 Assignment [tuple.assign]

For each tuple assignment operator, an exception is thrown only if the assignment of one of the types in Types throws an exception. In the function descriptions that follow, let i be in the range \([0, \text{sizeof...(Types)})\) in order, \(T_i\) be the \(i\)th type in Types, and \(U_i\) be the \(i\)th type in a template parameter pack named UTypes, where indexing is zero-based.

constexpr tuple& operator=(const tuple& u);

Effects: Assigns each element of u to the corresponding element of *this.

Returns: *this.

Remarks: This operator is defined as deleted unless is_copy_assignable_v<T_i> is true for all i.

costexpr tuple& operator=(tuple&& u) noexcept(see below);

Effects: For all i, assigns std::forward<T_i>(get<i>(u)) to get<i>(*this).

Returns: *this.

Remarks: The expression inside noexcept is equivalent to the logical AND of the following expressions:

is_nothrow_move_assignable_v<T_i>

where \(T_i\) is the \(i\)th type in Types.

template<class... UTypes> constexpr tuple& operator=(const tuple<UTypes...>& u);

Constraints:

(9.1) sizeof...(Types) equals sizeof...(UTypes) and
(9.2) is_assignable_v<T_i, const U_i&> is true for all i.

Effects: Assigns each element of u to the corresponding element of *this.

Returns: *this.

template<class... UTypes> constexpr tuple& operator=(tuple<UTypes...>&& u);

Constraints:

(12.1) sizeof...(Types) equals sizeof...(UTypes) and
(12.2) is_assignable_v<T_i, U_i> is true for all i.

Effects: For all i, assigns std::forward<U_i>(get<i>(u)) to get<i>(*this).

Returns: *this.

template<class U1, class U2> constexpr tuple& operator=(const pair<U1, U2>&& u);

Constraints:

(15.1) sizeof...(Types) is 2 and
(15.2) is_assignable_v<T_0, const U_1&> is true, and
(15.3) is_assignable_v<T_1, const U_2&> is true.
Effects: Assigns \texttt{u.first} to the first element of \texttt{*this} and \texttt{u.second} to the second element of \texttt{*this}.

Returns: \texttt{*this}.

\begin{verbatim}
template<class U1, class U2> constexpr tuple& operator=(pair<U1, U2>&& u);
\end{verbatim}

Constraints:
(18.1) sizeof...(Types) is 2 and
(18.2) is_assignable_v<T0&, U1> is true, and
(18.3) is_assignable_v<T1&, U2> is true.

Effects: Assigns \texttt{std::forward<U1>(u.first)} to the first element of \texttt{*this} and \texttt{std::forward<U2>(u.second)} to the second element of \texttt{*this}.

Returns: \texttt{*this}.

20.5.3.3 swap \[\text{ tuple.swap}\]

constexpr void swap(tuple& rhs) noexcept (see below);

Preconditions: Each element in \texttt{*this} is swappable with (16.4.4.3) the corresponding element in \texttt{rhs}.

Effects: Calls \texttt{swap} for each element in \texttt{*this} and its corresponding element in \texttt{rhs}.

Throws: Nothing unless one of the element-wise \texttt{swap} calls throws an exception.

Remarks: The expression inside \texttt{noexcept} is equivalent to the logical AND of the following expressions:
\texttt{is_nothrow_swappable_v<T_i>}
where \texttt{T_i} is the \texttt{i}th type in \texttt{Types}.

20.5.4 Tuple creation functions \[\text{ tuple.creation}\]

In the function descriptions that follow, the members of a template parameter pack \texttt{XTypes} are denoted by \texttt{X_i} for \texttt{i} in \([0, \text{sizeof...} (\text{XTypes}))\) in order, where indexing is zero-based.

\begin{verbatim}
template<class... TTypes>
constexpr tuple<unwrap_ref_decay_t<TTypes>...> make_tuple(TTypes&&... t);
\end{verbatim}

Returns: tuple<unwrap_ref_decay_t<TTypes>...>(std::forward<TTypes>(t)...).

\begin{example}
int i; float j;
make_tuple(1, ref(i), cref(j))
creates a tuple of type tuple<int, int&, const float&>. — end example
\end{example}

\begin{verbatim}
template<class... TTypes>
constexpr tuple<TTypes&&...> forward_as_tuple(TTypes&&... t) noexcept;
\end{verbatim}

Effects: Constructs a tuple of references to the arguments in \texttt{t} suitable for forwarding as arguments to a function. Because the result may contain references to temporary objects, a program shall ensure that the return value of this function does not outlive any of its arguments (e.g., the program should typically not store the result in a named variable).

Returns: tuple<TTypes&&...>(std::forward<TTypes>(t)...).

\begin{example}
tie functions allow one to create tuples that unpack tuples into variables. \texttt{ignore} can be used for elements that are not needed:
\begin{verbatim}
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
— end example
\end{verbatim}
\end{example}

§ 20.5.4 596
In the following paragraphs, let $T_i$ be the $i$th type in $Tuples$, $U_i$ be $\text{remove}\_\text{reference}\_t<T_i>$, and $tp_i$ be the $i$th parameter in the function parameter pack $tpls$, where all indexing is zero-based.

** Preconditions:** For all $i$, $U_i$ is the type $cv_i$tuple$<Args_i,...>$, where $cv_i$ is the (possibly empty) $i$th cv-qualifier-seq and $Args_i$ is the template parameter pack representing the element types in $U_i$. Let $A_{ik}$ be the $k$th type in $Args_i$. For all $A_{ik}$ the following requirements are met:

9.1 If $T_i$ is deduced as an lvalue reference type, then $\text{is}\_\text{constructible}\_v<A_{ik}, cv_i A_{ik}> == true$, otherwise

9.2 $\text{is}\_\text{constructible}\_v<A_{ik}, cv_i A_{ik}&&> == true$.

** Remarks:** The types in $CTypes$ are equal to the ordered sequence of the extended types $Args_0,..., Args_1,..., Args_{n-1}$, where $n$ is equal to $\text{sizeof}(...Tuples)$. Let $e_i,...$ be the $i$th ordered sequence of tuple elements of the resulting tuple object corresponding to the type sequence $Args_i$.

** Returns:** A tuple object constructed by initializing the $k_i$th type element $e_{ik}$ in $e_i,...$ with $\text{get}<k_i>(\text{std}::\text{forward}<T_i>(tp_i))$ for each valid $k_i$ and each group $e_i$ in order.

[Note 1: An implementation can support additional types in the template parameter pack $Tuples$ that support the tuple-like protocol, such as pair and array. —end note]

### 20.5.5 Calling a function with a tuple of arguments [tuple.apply]

```cpp
template<class F, class Tuple>
constexpr decltype(auto) apply(F&& f, Tuple&& t);
```

** Effects:** Given the exposition-only function:

```cpp
template<class F, class Tuple, size_t... I>
constexpr decltype(auto) apply-impl(F&& f, Tuple&& t, index_sequence<I...>) {
    // exposition only
    return INVOKE(\text{std}::\text{forward}<F>(f), \text{std}::\text{get}<I>(\text{std}::\text{forward}<\text{Tuple}>(t))...); // see 20.14.4
}
```

Equivalent to:

```cpp
return apply-impl(\text{std}::\text{forward}<F>(f), \text{std}::\text{forward}<\text{Tuple}>(t),
    \text{make\_index\_sequence}<\text{tuple\_size\_v}<\text{remove}\_\text{reference}\_t<\text{Tuple}>>{}){
}
```

```cpp
template<class T, class Tuple>
constexpr T make_from_tuple(Tuple&& t);
```

** Effects:** Given the exposition-only function:

```cpp
template<class T, class Tuple, size_t... I>
constexpr T make-from-tuple-impl(Tuple&& t, index_sequence<I...>) { // exposition only
    return T(\text{get}<I>(\text{std}::\text{forward}<\text{Tuple}>(t))...);  
}
```

Equivalent to:

```cpp
return make-from-tuple-impl<T>(
    forward<\text{Tuple}>(t),
    \text{make\_index\_sequence}<\text{tuple\_size\_v}<\text{remove}\_\text{reference}\_t<\text{Tuple}>>{}){
}
```

[Note 1: The type of T must be supplied as an explicit template parameter, as it cannot be deduced from the argument list. —end note]

### 20.5.6 Tuple helper classes [tuple.helper]

```cpp
template<class T> struct tuple_size;
```

All specializations of $\text{tuple}\_\text{size}$ meet the $\text{Cpp17UnaryTypeTrait}$ requirements (20.15.2) with a base characteristic of $\text{integral}\_\text{constant}<\text{size}_t, N>$ for some $N$.

```cpp
template<class... Types>
struct tuple_size<Types...> : public integral_constant<\text{size}_t, \text{sizeof}(...(Types)> { };  
```
template<size_t I, class... Types>
struct tuple_element<I, tuple<Types...>> {
  using type = TI;
};

Mandates: $I < \text{sizeof...}(\text{Types})$.

Type: TI is the type of the $I$th element of Types, where indexing is zero-based.

template<class T> struct tuple_size<const T>;

Let TS denote $\text{tuple_size}<T>$ of the cv-unqualified type T. If the expression $\text{TS}::\text{value}$ is well-formed when treated as an unevaluated operand, then each specialization of the template meets the Cpp17Unary-TypeTrait requirements (20.15.2) with a base characteristic of

\[
\text{integral_constant}<\text{size_t}, \text{TS}::\text{value}>
\]

Otherwise, it has no member value.

Access checking is performed as if in a context unrelated to TS and T. Only the validity of the immediate context of the expression is considered.

[Note 1: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. — end note]

In addition to being available via inclusion of the <tuple> header, the template is available when any of the headers <array> (22.3.2), <ranges> (24.2), or <utility> (20.2.1) are included.

template<size_t I, class T> struct tuple_element<I, const T>;

Let TE denote $\text{tuple_element_t}<I, T>$ of the cv-unqualified type T. Then each specialization of the template meets the Cpp17TransformationTrait requirements (20.15.2) with a member typedef $\text{type}$ that names the type $\text{add_const_t}<\text{TE}>$.

In addition to being available via inclusion of the <tuple> header, the template is available when any of the headers <array> (22.3.2), <ranges> (24.2), or <utility> (20.2.1) are included.

20.5.7 Element access

\[
\begin{align*}
\text{template}\langle\text{size_t I, class... Types}\rangle & \quad \text{constexpr tuple_element_t<I, tuple<Types...>>& get(tuple<Types...>& t) noexcept;} \\
\text{template}\langle\text{size_t I, class... Types}\rangle & \quad \text{constexpr tuple_element_t<I, tuple<Types...>>&& get(tuple<Types...>&& t) noexcept; } \quad \text{// Note A} \\
\text{template}\langle\text{size_t I, class... Types}\rangle & \quad \text{constexpr const tuple_element_t<I, tuple<Types...>>& get(const tuple<Types...>& t) noexcept; } \quad \text{// Note B} \\
\text{template}\langle\text{size_t I, class... Types}\rangle & \quad \text{constexpr const tuple_element_t<I, tuple<Types...>>&& get(const tuple<Types...>&& t) noexcept;} \\
\end{align*}
\]

Mandates: $I < \text{sizeof...}(\text{Types})$.

Returns: A reference to the $I$th element of $t$, where indexing is zero-based.

[Note 1: [Note A] If a type $T$ in $\text{Types}$ is some reference type $X&$, the return type is $X&$, not $X&&$. However, if the element type is a non-reference type $T$, the return type is $T&$. — end note]

[Note 2: [Note B] Constness is shallow. If a type $T$ in $\text{Types}$ is some reference type $X&$, the return type is $X&$, not $\text{const X}&$. However, if the element type is a non-reference type $T$, the return type is $\text{const T}&$. This is consistent with how constness is defined to work for member variables of reference type. — end note]
template<class T, class... Types>
constexpr const T&& get(const tuple<Types...>&& t) noexcept;

Mandates: The type T occurs exactly once in Types.

Returns: A reference to the element of t corresponding to the type T in Types.

[Example 1:]
const tuple<int, const int, double, double> t(1, 2, 3.4, 5.6);
const int& i1 = get<int>(t);  // OK, i1 has value 1
const int& i2 = get<const int>(t);  // OK, i2 has value 2
const double& d = get<double>(t);  // error: type double is not unique within t
—end example]

Note 3: The reason get is a non-member function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the template keyword. —end note]

20.5.8 Relational operators [tuple.rel]

template<class... TTypes, class... UTypes>
constexpr bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

Mandates: For all i, where 0 ≤ i < sizeof...(TTypes), get<i>(t) == get<i>(u) is a valid expression returning a type that is convertible to bool. sizeof...(TTypes) equals sizeof...(UTypes).

Returns: true if get<i>(t) == get<i>(u) for all i, otherwise false. For any two zero-length tuples e and f, e == f returns true.

Remarks: The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to false.

template<class... TTypes, class... UTypes>
constexpr common_comparison_category_t<synth-three-way-result<TTypes, UTypes>...> operator<=>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

Effects: Performs a lexicographical comparison between t and u. For any two zero-length tuples t and u, t <=> u returns strong_ordering::equal. Otherwise, equivalent to:

if (auto c = synth-three-way(get<0>(t), get<0>(u)); c != 0) return c;
return t.tail <=> u.tail;

where r.tail for some tuple r is a tuple containing all but the first element of r.

Note 1: The above definition does not require t.tail (or u.tail) to be constructed. It might not even be possible, as t and u are not required to be copy constructible. Also, all comparison operator functions are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. —end note]

20.5.9 Tuple traits [tuple.traits]

template<class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc> : true_type {
};

Preconditions: Alloc meets the Cpp17Allocator requirements (Table 36).

[Note 1: Specialization of this trait informs other library components that tuple can be constructed with an allocator, even though it does not have a nested allocator_type. —end note]

20.5.10 Tuple specialized algorithms [tuple.special]

template<class... Types>
constexpr void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);

Constraints: is_swappable_v<T> is true for every type T in Types.

Effects: As if by x.swap(y).

Remarks: The expression inside noexcept is equivalent to:

noexcept(x.swap(y))
20.6 Optional objects

20.6.1 In general

Subclause 20.6 describes class template `optional` that represents optional objects. An optional object is an object that contains the storage for another object and manages the lifetime of this contained object, if any. The contained object may be initialized after the optional object has been initialized, and may be destroyed before the optional object has been destroyed. The initialization state of the contained object is tracked by the optional object.

20.6.2 Header `<optional>` synopsis

```
#include <compare>  // see 17.11.1

namespace std {
    // 20.6.3, class template optional
    template<class T>
    class optional;

    // 20.6.4, no-value state indicator
    struct nullopt_t {
        see below
    };
    inline constexpr nullopt_t nullopt(unsspecified);

    // 20.6.5, class bad_optional_access
    class bad_optional_access;

    // 20.6.6, relational operators
    template<class T, class U>
    constexpr bool operator==(const optional<T>&, const optional<U>&);
    template<class T, class U>
    constexpr bool operator!=(const optional<T>&, const optional<U>&);
    template<class T, class U>
    constexpr bool operator<(const optional<T>&, const optional<U>&);
    template<class T, class U>
    constexpr bool operator>(const optional<T>&, const optional<U>&);
    template<class T, class U>
    constexpr bool operator<=(const optional<T>&, const optional<U>&);
    template<class T, class U>
    constexpr bool operator>=(const optional<T>&, const optional<U>&);
    template<class T, three_way_comparable_with<T> U>
    constexpr compare_three_way_result_t<T,U> operator<=>(const optional<T>&, const optional<U>&);

    // 20.6.7, comparison with nullopt
    template<class T> constexpr bool operator==(const optional<T>&, nullopt_t) noexcept;
    template<class T> constexpr strong_ordering operator<=>(const optional<T>&, nullopt_t) noexcept;

    // 20.6.8, comparison with T
    template<class T, class U> constexpr bool operator==(const optional<T>&, const U&);
    template<class T, class U> constexpr bool operator==(const T&, const optional<U>&);
    template<class T, class U> constexpr bool operator!=(const optional<T>&, const U&);
    template<class T, class U> constexpr bool operator!=(const T&, const optional<U>&);
    template<class T, class U> constexpr bool operator<(const optional<T>&, const U&);
    template<class T, class U> constexpr bool operator<(const T&, const optional<U>&);
    template<class T, class U> constexpr bool operator>(const optional<T>&, const U&);
    template<class T, class U> constexpr bool operator>(const T&, const optional<U>&);
    template<class T, class U> constexpr bool operator<=(const optional<T>&, const U&);
    template<class T, class U> constexpr bool operator<=(const T&, const optional<U>&);
    template<class T, class U> constexpr bool operator>=(const optional<T>&, const U&);
    template<class T, class U> constexpr bool operator>=(const T&, const optional<U>&);
    template<class T, three_way_comparable_with<T> U>
    constexpr compare_three_way_result_t<T,U> operator<=>(const optional<T>&, const U&);
}
```
// 20.6.9, specialized algorithms

```cpp
// 20.6.10, hash support

template<class T>
constexpr optional<T> make_optional(T&&);
```

```cpp
template<class T, class... Args>
constexpr optional<T> make_optional(initializer_list<Args> il, Args&&... args);
```

### 20.6.3 Class template optional

#### 20.6.3.1 General

```cpp
namespace std {
  template<class T>
  class optional {
    public:
      using value_type = T;

    // 20.6.3.2, constructors
    constexpr optional() noexcept;
    constexpr optional(nullopt_t) noexcept;
    constexpr optional(const optional&);
    constexpr optional(optional&&) noexcept;
    template<class... Args>
      constexpr explicit optional(in_place_t, Args&&...);
    template<class U, class... Args>
      constexpr explicit optional(in_place_t, initializer_list<U>, Args&&...);
    template<class U = T>
      constexpr explicit(optional(U&&));
    template<class U>
      explicit(optional(const optional<U>&));
    template<class U>
      explicit(optional(optional<U>&&));

    // 20.6.3.3, destructor
    ~optional();

    // 20.6.3.4, assignment
    optional& operator=(nullopt_t) noexcept;
    constexpr optional& operator=(const optional&);
    constexpr optional& operator=(optional&&) noexcept;
    template<class U = T>
      optional& operator=(optional<U&&>);
    template<class U>
      explicit(optional(const optional<U>&));
    template<class U>
      explicit(optional(optional<U>&&));
    template<class... Args>
      T& emplace(initializer_list<Args>, Args&&...);

    // 20.6.3.5, swap
    void swap(optional&) noexcept;

    // 20.6.3.6, observers
    constexpr const T* operator->() const;
    constexpr T* operator->();
    constexpr const T& operator*() const&;
    constexpr T& operator*() &;
    constexpr T&& operator*() &&;
    constexpr const T&& operator*() const&&;
```
constexpr explicit operator bool() const noexcept;
constexpr bool has_value() const noexcept;
constexpr const T& value() const&;
constexpr T& value() &;
constexpr T&& value() &&;
constexpr const T&& value() const&&;

// 20.6.3.7, modifiers
void reset() noexcept;

private:
 T *val;     // exposition only
};

template<class T>
optional(T) -> optional<T>;

Any instance of optional<T> at any given time either contains a value or does not contain a value. When an instance of optional<T> contains a value, it means that an object of type T, referred to as the optional object’s contained value, is allocated within the storage of the optional object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained value. The contained value shall be allocated in a region of the optional<T> storage suitably aligned for the type T. When an object of type optional<T> is contextually converted to bool, the conversion returns true if the object contains a value; otherwise the conversion returns false.

Member val is provided for exposition only. When an optional<T> object contains a value, val points to the contained value.

T shall be a type other than cv in_place_t or cv nullopt_t that meets the Cpp17Destructible requirements (Table 32).

20.6.3.2 Constructors

constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;

1 Postconditions: *this does not contain a value.
2 Remarks: No contained value is initialized. For every object type T these constructors areconstexpr constructors (9.2.6).

constexpr optional(const optional& rhs);
3 Effects: If rhs contains a value, initializes the contained value as if direct-non-list-initializing an object of type T with the expression *rhs.
4 Postconditions: bool(rhs) == bool(*this).
5 Throws: Any exception thrown by the selected constructor of T.
6 Remarks: This constructor is defined as deleted unless is_copy_constructible_v<T> is true. If is_trivially_copy_constructible_v<T> is true, this constructor is trivial.

constexpr optional(optional&& rhs) noexcept(see below);
7 Constraints: is_move_constructible_v<T> is true.
8 Effects: If rhs contains a value, initializes the contained value as if direct-non-list-initializing an object of type T with the expression std::move(*rhs). bool(rhs) is unchanged.
9 Postconditions: bool(rhs) == bool(*this).
10 Throws: Any exception thrown by the selected constructor of T.
11 Remarks: The expression inside noexcept is equivalent to is_nothrow_move_constructible_v<T>. If is_trivially_move_constructible_v<T> is true, this constructor is trivial.

§ 20.6.3.2
template<class... Args> constexpr explicit optional(in_place_t, Args&&... args);

Constraints: is_constructible_v<T, Args...> is true.
Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments std::forward<Args>(args)....
Postconditions: *this contains a value.
Throws: Any exception thrown by the selected constructor of T.
Remarks: If T’s constructor selected for the initialization is a constexpr constructor, this constructor is a constexpr constructor.

template<class U, class... Args>
constexpr explicit optional(in_place_t, initializer_list<U> il, Args&&... args);

Constraints: is_constructible_v<T, initializer_list<U>&, Args...> is true.
Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments il, std::forward<Args>(args)....
Postconditions: *this contains a value.
Throws: Any exception thrown by the selected constructor of T.
Remarks: If T’s constructor selected for the initialization is a constexpr constructor, this constructor is a constexpr constructor.

template<class U = T> constexpr explicit(U&& v);

Constraints: is_constructible_v<T, U> is true, is_same_v<remove_cvref_t<U>, in_place_t> is false, and is_same_v<remove_cvref_t<U>, optional> is false.
Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the expression std::forward<U>(v).
Postconditions: *this contains a value.
Throws: Any exception thrown by the selected constructor of T.
Remarks: If T’s selected constructor is a constexpr constructor, this constructor is a constexpr constructor. The expression inside explicit is equivalent to:

!is_convertible_v<U, T>

template<class U> explicit(const optional<U>& rhs);

Constraints:
(27.1) is_constructible_v<T, const U&> is true,
(27.2) is_constructible_v<T, optional<U>&> is false,
(27.3) is_constructible_v<T, optional<U>&&> is false,
(27.4) is_constructible_v<T, const optional<U>&> is false,
(27.5) is_constructible_v<T, const optional<U>&&> is false,
(27.6) is_convertible_v<optional<U>&, T> is false,
(27.7) is_convertible_v<optional<U>&& , T> is false,
(27.8) is_convertible_v<const optional<U>&, T> is false, and
(27.9) is_convertible_v<const optional<U>&& , T> is false.
Effects: If rhs contains a value, initializes the contained value as if direct-non-list-initializing an object of type T with the expression *rhs.
Postconditions: bool(rhs) == bool(*this).
Throws: Any exception thrown by the selected constructor of T.
Remarks: The expression inside explicit is equivalent to:

!is_convertible_v<const U&, T>
template<class U> explicit(see below) optional(optional<U>&& rhs);

Constraints:

(32.1) \(\text{is\_constructible\_v<T, U>}\) is true,
(32.2) \(\text{is\_constructible\_v<T, optional<U>&>}\) is false,
(32.3) \(\text{is\_constructible\_v<T, optional<U>&&>}\) is false,
(32.4) \(\text{is\_constructible\_v<T, const optional<U>&>}\) is false,
(32.5) \(\text{is\_constructible\_v<T, const optional<U>&&>}\) is false,
(32.6) \(\text{is\_convertible\_v<optional<U>&, T>}\) is false,
(32.7) \(\text{is\_convertible\_v<optional<U>&&, T>}\) is false,
(32.8) \(\text{is\_convertible\_v<const optional<U>&, T>}\) is false, and
(32.9) \(\text{is\_convertible\_v<const optional<U>&&, T>}\) is false.

Effects: If \(\text{rhs}\) contains a value, initializes the contained value as if direct-non-list-initializing an object of type \(T\) with the expression \(\text{std::move(*rhs)}\). \(\text{bool(rhs)}\) is unchanged.

Postconditions: \(\text{bool(rhs)} == \text{bool(*this)}\).

Throws: Any exception thrown by the selected constructor of \(T\).

Remarks: The expression inside \texttt{explicit} is equivalent to:

\[
\text{!is\_convertible\_v<U, T>}
\]

20.6.3.3 Destructor

\text{~optional();}

Effects: If \(\text{is\_trivially\_destructible\_v<T>! = true}\) and \(\text{*this}\) contains a value, calls \(\text{val->T::~T()}\).

Remarks: If \(\text{is\_trivially\_destructible\_v<T>}\) is true, then this destructor is trivial.

20.6.3.4 Assignment

\texttt{optional<T>& operator=(nullopt_t) noexcept;}

Effects: If \(\text{*this}\) contains a value, calls \(\text{val->T::~T()}\) to destroy the contained value; otherwise no effect.

Postconditions: \(\text{*this}\) does not contain a value.

Returns: \(\text{*this}\).

\text{constexpr optional<T>& operator=(const optional& rhs);}

Effects: See Table 42.

Table 42: \texttt{optional::operator=(const optional&)} effects

<table>
<thead>
<tr>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{rhs contains a value})</td>
<td>assigns (\text{rhs}) to the contained value</td>
</tr>
<tr>
<td>(\text{rhs does not contain a value})</td>
<td>destroys the contained value by calling (\text{val-&gt;T::~T()})</td>
</tr>
</tbody>
</table>

Postconditions: \(\text{bool(rhs)} == \text{bool(*this)}\).

Returns: \(\text{*this}\).

Remarks: If any exception is thrown, the result of the expression \(\text{bool(*this)}\) remains unchanged. If an exception is thrown during the call to \(T\)'s copy constructor, no effect. If an exception is thrown during the call to \(T\)'s copy assignment, the state of its contained value is as defined by the exception safety guarantee.
of T's copy assignment. This operator is defined as deleted unless is_copy_constructible_v<T> is true and is_copy_assignable_v<T> is true. If is_trivially_copy_constructible_v<T> && is_trivially_copy_assignable_v<T> && is_trivially_destructible_v<T> is true, this assignment operator is trivial.

constexpr optional& operator=(optional&& rhs) noexcept(see below);

Constraints: is_move_constructible_v<T> is true and is_move_assignable_v<T> is true.

Effects: See Table 43. The result of the expression bool(rhs) remains unchanged.

Table 43: optional::operator=(optional&&) effects

<table>
<thead>
<tr>
<th></th>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs contains a value</td>
<td>assigns std::move(*rhs) to the contained value</td>
<td>initializes the contained value as if direct-non-list-initializing an object of type T with std::move(*rhs)</td>
</tr>
<tr>
<td>rhs does not contain a value</td>
<td>destroys the contained value by calling val-&gt;T::~T()</td>
<td>no effect</td>
</tr>
</tbody>
</table>

Postconditions: bool(rhs) == bool(*this).

Returns: *this.

Remarks: The expression inside noexcept is equivalent to:

  is_nothrow_move_assignable_v<T> && is_nothrow_move_constructible_v<T>

If any exception is thrown, the result of the expression bool(*this) remains unchanged. If an exception is thrown during the call to T's move constructor, the state of *rhs.val is determined by the exception safety guarantee of T's move constructor. If an exception is thrown during the call to T's move assignment, the state of *val and *rhs.val is determined by the exception safety guarantee of T's move assignment. If is_trivially_move_constructible_v<T> && is_trivially_move_assignable_v<T> && is_trivially_destructible_v<T> is true, this assignment operator is trivial.

template<class U = T> optional<T>& operator=(U&& v);

Constraints:

(19.1) is_constructible_v<T, const U&> is true,
(19.2) is_assignable_v<T&, const U&> is true,
(19.3) is_constructible_v<T, optional<U>&> is false,
(19.4) is_constructible_v<T, optional<U>&&> is false,
(19.5) is_constructible_v<T, const optional<U>&> is false,
(19.6) is_constructible_v<T, const optional<U>&&> is false,
is_convertible_v<optional<U>&, T> is false,
— is_convertible_v<optional<U>&&, T> is false,
— is_convertible_v<const optional<U>&, T> is false,
— is_convertible_v<const optional<U>&&, T> is false,
— is_assignable_v<T&, optional<U>&> is false,
— is_assignable_v<T&, optional<U>&&> is false,
— is_assignable_v<T&, const optional<U>&> is false, and
— is_assignable_v<T&, const optional<U>&&> is false.

Effects: See Table 44.

Table 44: optional::operator=(const optional<U>&) effects

<table>
<thead>
<tr>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs contains a value</td>
<td>assigns *rhs to the contained value</td>
</tr>
<tr>
<td>rhs does not contain a value</td>
<td>destroys the contained value by calling val-&gt;T::~T()</td>
</tr>
<tr>
<td></td>
<td>initializes the contained value as if direct-non-list-initializing an object of type T with *rhs</td>
</tr>
<tr>
<td></td>
<td>no effect</td>
</tr>
</tbody>
</table>

Postconditions: bool(rhs) == bool(*this).

Returns: *this.

Remarks: If any exception is thrown, the result of the expression bool(*this) remains unchanged. If an exception is thrown during the call to T’s constructor, the state of *rhs.val is determined by the exception safety guarantee of T’s constructor. If an exception is thrown during the call to T’s assignment, the state of *val and *rhs.val is determined by the exception safety guarantee of T’s assignment.

template<class U> optional<T>& operator=(optional<U>&& rhs);

Constraints:
— is_constructible_v<T, U> is true,
— is_assignable_v<T&, U> is true,
— is_constructible_v<T, optional<U>&> is false,
— is_constructible_v<T, optional<U>&&> is false,
— is_constructible_v<T, const optional<U>&> is false,
— is_constructible_v<T, const optional<U>&&> is false,
— is_convertible_v<optional<U>&, T> is false,
— is_convertible_v<optional<U>&&, T> is false,
— is_convertible_v<optional<U>&, T> is false,
— is_convertible_v<optional<U>&& , T> is false,
— is_assignable_v<T&, optional<U>&> is false,
— is_assignable_v<T&, optional<U>&&> is false,
— is_assignable_v<T&, const optional<U>&> is false, and
— is_assignable_v<T&, const optional<U>&&> is false.

Effects: See Table 45. The result of the expression bool(rhs) remains unchanged.

Postconditions: bool(rhs) == bool(*this).

Returns: *this.
Table 45: optional::operator=(optional<U>&&) effects

<table>
<thead>
<tr>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rhs contains a value</strong></td>
<td>assigns std::move(*rhs) to the contained value</td>
</tr>
<tr>
<td><strong>rhs does not contain a value</strong></td>
<td>destroys the contained value by calling val-&gt;T::~T()</td>
</tr>
</tbody>
</table>

Remarks: If any exception is thrown, the result of the expression bool(*this) remains unchanged. If an exception is thrown during the call to T's constructor, the state of *rhs.val is determined by the exception safety guarantee of T's constructor. If an exception is thrown during the call to T's assignment, the state of *val and *rhs.val is determined by the exception safety guarantee of T's assignment.

template<class... Args> T& emplace(Args&&... args);

Mandates: is_constructible_v<T, Args...> is true.

Effects: Calls *this = nullopt. Then initializes the contained value as if direct-non-list-initializing an object of type T with the arguments std::forward<Args>(args)....

Postconditions: *this contains a value.

Returns: A reference to the new contained value.

Remarks: If an exception is thrown during the call to T's constructor, *this does not contain a value, and the previous *val (if any) has been destroyed.

template<class U, class... Args> T& emplace(initializer_list<U> il, Args&&... args);

Constraints: is_constructible_v<T, initializer_list<U>&, Args...> is true.

Effects: Calls *this = nullopt. Then initializes the contained value as if direct-non-list-initializing an object of type T with the arguments il, std::forward<Args>(args)....

Postconditions: *this contains a value.

Returns: A reference to the new contained value.

Remarks: If an exception is thrown by the selected constructor of T.

Swap

void swap(optional& rhs) noexcept(see below);

Mandates: is_move_constructible_v<T> is true.

Preconditions: Lvalues of type T are swappable.

Effects: See Table 46.

Throws: Any exceptions thrown by the operations in the relevant part of Table 46.

Remarks: The expression inside noexcept is equivalent to:

```cpp
is_nothrow_move_constructible_v<T> && is_nothrow_swappable_v<T>
```

If any exception is thrown, the results of the expressions bool(*this) and bool(rhs) remain unchanged. If an exception is thrown during the call to function swap, the state of *val and *rhs.val is determined by the exception safety guarantee of swap for lvalues of T. If an exception is thrown during the call to T's move constructor, the state of *val and *rhs.val is determined by the exception safety guarantee of T's move constructor.
### Table 46: `optional::swap(optional&)` effects

<table>
<thead>
<tr>
<th></th>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rhs contains a value</strong></td>
<td>- calls <code>swap(*(*this), *rhs)</code></td>
<td>- initializes the contained value of *this as if direct-non-list-initializing an object of type T with the expression <code>std::move(*rhs)</code>, followed by <code>rhs.val-&gt;T::~T()</code>; postcondition is that *this contains a value and rhs does not contain a value</td>
</tr>
<tr>
<td><strong>rhs does not contain a value</strong></td>
<td>- initializes the contained value of rhs as if direct-non-list-initializing an object of type T with the expression <code>std::move(*(*this))</code>, followed by <code>val-&gt;T::~T()</code>; postcondition is that *this does not contain a value and rhs contains a value</td>
<td>- no effect</td>
</tr>
</tbody>
</table>

### 20.6.3.6 Observers

```cpp
constexpr const T* operator->() const;
constexpr T* operator->();
```

1. **Preconditions:** *this contains a value.
2. **Returns:** val.
3. **Throws:** Nothing.
4. **Remarks:** These functions are constexpr functions.

```cpp
constexpr const T& operator*() const&;
constexpr T& operator*() &;
```

5. **Preconditions:** *this contains a value.
6. **Returns:** *val.
7. **Throws:** Nothing.
8. **Remarks:** These functions are constexpr functions.

```cpp
constexpr T&& operator*() &&;
constexpr const T&& operator*() const&&;
```

9. **Preconditions:** *this contains a value.
10. **Effects:** Equivalent to: return `std::move(*val)`;
11. **Returns:** true if and only if *this contains a value.
12. **Remarks:** This function is a constexpr function.

```cpp
constexpr explicit operator bool() const noexcept;
```

13. **Returns:** true if and only if *this contains a value.
14. **Remarks:** This function is a constexpr function.

```cpp
constexpr const T& value() const&;
```

§ 20.6.3.6
constexpr T& value() &;
\textit{Effects:} Equivalent to:
\[
\text{return bool(*this) ? *val : throw bad_optional_access();}
\]
constexpr T&& value() &&;
constexpr const T&& value() const&&;
\textit{Effects:} Equivalent to:
\[
\text{return bool(*this) ? std::move(*val) : throw bad_optional_access();}
\]
\text{template<class U> constexpr T value_or(U&& v) const&;}
\textit{Mandates:} is_copy_constructible_v<T> && is_convertible_v<U&&, T> is true.
\textit{Effects:} Equivalent to:
\[
\text{return bool(*this) ? **this : static_cast<T>(std::forward<U>(v));}
\]\n\text{template<class U> constexpr T value_or(U&& v) &&;}
\textit{Mandates:} is_move_constructible_v<T> && is_convertible_v<U&&, T> is true.
\textit{Effects:} Equivalent to:
\[
\text{return bool(*this) ? std::move(**this) : static_cast<T>(std::forward<U>(v));}
\]
\section{Modifiers}
\text{[optional.mod]}
void reset() noexcept;
\textit{Effects:} If \texttt{*this} contains a value, calls \texttt{val->T::~T()} to destroy the contained value; otherwise no effect.
\textit{Postconditions:} \texttt{*this} does not contain a value.
\section{No-value state indicator}
\text{[optional.nullopt]}
\text{struct nullopt_t{\textit{see below}};}
\text{inline constexpr nullopt_t nullopt(unspecified);}
\text{The struct \texttt{nullopt_t} is an empty class type used as a unique type to indicate the state of not containing a value for \texttt{optional} objects. In particular, \texttt{optional<T>} has a constructor with \texttt{nullopt_t} as a single argument; this indicates that an optional object not containing a value shall be constructed.}
\text{Type \texttt{nullopt_t} shall not have a default constructor or an initializer-list constructor, and shall not be an aggregate.}
\section{Class \texttt{bad\_optional\_access}}
\text{[optional.bad.access]}
\text{class bad\_optional\_access : public exception {
public:
\textit{// see 17.9.3 for the specification of the special member functions}
\textit{const char\* what() const noexcept override;}
\};}
\text{The class \texttt{bad\_optional\_access} defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the value of an optional object that does not contain a value.}
\text{const char\* what() const noexcept override;
\textit{Returns:} An implementation-defined \texttt{ntbs}.}
\section{Relational operators}
\text{[optional.relops]}
\text{template<class T, class U> constexpr bool operator==(const optional<T>& x, const optional<U>& y);}
\textit{Mandates:} The expression \texttt{*x == *y} is well-formed and its result is convertible to \texttt{bool}.
\textit{[Note 1: T need not be Cpp17EqualityComparable. — end note]}
\textit{Returns:} If \texttt{bool(x) \!= bool(y)}, \texttt{false}; otherwise if \texttt{bool(x) \== false}, \texttt{true}; otherwise \texttt{*x \== *y}.
\textit{Remarks:} Specializations of this function template for which \texttt{*x \== *y} is a core constant expression are constexpr functions.
template<class T, class U> constexpr bool operator!=(const optional<T>& x, const optional<U>& y);

Mandates: The expression *x != *y is well-formed and its result is convertible to bool.

Returns: If bool(x) != bool(y), true; otherwise, if bool(x) == false, false; otherwise *x != *y.

Remarks: Specializations of this function template for which *x != *y is a core constant expression are constexpr functions.

template<class T, class U> constexpr bool operator<(const optional<T>& x, const optional<U>& y);

Mandates: *x < *y is well-formed and its result is convertible to bool.

Returns: If !y, false; otherwise, if !x, true; otherwise *x < *y.

Remarks: Specializations of this function template for which *x < *y is a core constant expression are constexpr functions.

template<class T, class U> constexpr bool operator>(const optional<T>& x, const optional<U>& y);

Mandates: The expression *x > *y is well-formed and its result is convertible to bool.

Returns: If !x, false; otherwise, if !y, true; otherwise *x > *y.

Remarks: Specializations of this function template for which *x > *y is a core constant expression are constexpr functions.

template<class T, class U> constexpr bool operator<=(const optional<T>& x, const optional<U>& y);

Mandates: The expression *x <= *y is well-formed and its result is convertible to bool.

Returns: If !x, true; otherwise, if !y, false; otherwise *x <= *y.

Remarks: Specializations of this function template for which *x <= *y is a core constant expression are constexpr functions.

template<class T, class U> constexpr bool operator>=(const optional<T>& x, const optional<U>& y);

Mandates: The expression *x >= *y is well-formed and its result is convertible to bool.

Returns: If !y, true; otherwise, if !x, false; otherwise *x >= *y.

Remarks: Specializations of this function template for which *x >= *y is a core constant expression are constexpr functions.

template<class T, class U> constexpr compare_three_way_result_t<T,U> operator<=>(const optional<T>& x, const optional<U>& y);

Returns: If x && y, *x <=> *y; otherwise bool(x) <=> bool(y).

Remarks: Specializations of this function template for which *x <=> *y is a core constant expression are constexpr functions.

20.6.7 Comparison with nullopt

template<class T> constexpr bool operator==(const optional<T>& x, nullopt_t) noexcept;

Returns: !x.

template<class T> constexpr strong_ordering operator<=>(const optional<T>& x, nullopt_t) noexcept;

Returns: bool(x) <=> false.

20.6.8 Comparison with T

template<class T, class U> constexpr bool operator==(const optional<T>& x, const optional<U>& y);

Mandates: The expression *x == v is well-formed and its result is convertible to bool.

[Note 1: T need not be Cpp17EqualityComparable. — end note]

Effects: Equivalent to: return bool(x) ? *x == v : false;

template<class T, class U> constexpr bool operator==(const T& x, const optional<U>& y);

Mandates: The expression v == *x is well-formed and its result is convertible to bool.
4. Effects: Equivalent to: return bool(x) ? v == *x : false;

template<class T, class U> constexpr bool operator!=(const optional<T>& x, const U& v);

Mandates: The expression *x != v is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? *x != v : true;

5. template<class T, class U> constexpr bool operator!=(const T& v, const optional<U>& x);

Mandates: The expression v != *x is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? v != *x : true;

6. template<class T, class U> constexpr bool operator<(const optional<T>& x, const U& v);

Mandates: The expression *x < v is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? *x < v : true;

7. template<class T, class U> constexpr bool operator<(const T& v, const optional<U>& x);

Mandates: The expression v < *x is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? v < *x : false;

8. template<class T, class U> constexpr bool operator>(const optional<T>& x, const U& v);

Mandates: The expression *x > v is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? *x > v : true;

9. template<class T, class U> constexpr bool operator>(const T& v, const optional<U>& x);

Mandates: The expression v > *x is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? v > *x : false;

10. template<class T, class U> constexpr bool operator<=(const optional<T>& x, const U& v);

Mandates: The expression *x <= v is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? *x <= v : true;

11. template<class T, class U> constexpr bool operator<=(const T& v, const optional<U>& x);

Mandates: The expression v <= *x is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? v <= *x : false;

12. template<class T, class U> constexpr bool operator>=(const optional<T>& x, const U& v);

Mandates: The expression *x >= v is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? *x >= v : false;

13. template<class T, class U> constexpr bool operator>=(const T& v, const optional<U>& x);

Mandates: The expression v >= *x is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? v >= *x : true;

14. template<class T, three_way_comparable_with<T> U> constexpr compare_three_way_result_t<T,U> operator<=>(const optional<T>& x, const U& v);

Mandates: The expression *x <=> v is well-formed and its result is convertible to bool.

Effects: Equivalent to: return bool(x) ? *x <=> v : strong_ordering::less;
template<class T, class... Args>
constexpr optional<T> make_optional(Args&&... args);

Effects: Equivalent to: return optional<T>(in_place, std::forward<Args>(args)...);

template<class T, class U, class... Args>
constexpr optional<T> make_optional(initializer_list<U> il, Args&&... args);

Effects: Equivalent to: return optional<T>(in_place, il, std::forward<Args>(args)...);

20.6.10 Hash support

template<class T> struct hash<optional<T>>;

The specialization hash<optional<T>> is enabled (20.14.19) if and only if hash<remove_const_<
T<T>> is enabled. When enabled, for an object o of type optional<T>, if bool(o) == true, then
hash<optional<T>>()(o) evaluates to the same value as hash<remove_const_t<T>>()(*o); otherwise
it evaluates to an unspecified value. The member functions are not guaranteed to be noexcept.

20.7 Variants

20.7.1 In general

A variant object holds and manages the lifetime of a value. If the variant holds a value, that value's
type has to be one of the template argument types given to variant. These template arguments are called
alternatives.

20.7.2 Header <variant> synopsis

#include <compare>  // see 17.11.1

namespace std {
    // 20.7.3, class template variant
template<class... Types>
    class variant;

    // 20.7.4, variant helper classes
template<class T> struct variant_size;  // not defined
    template<class T> struct variant_size<const T>;
    template<class T>
        inline constexpr size_t variant_size_v = variant_size<T>::value;

    template<class... Types>
        struct variant_size<variant<Types...>>;

    template<size_t I, class... Types>
        struct variant_alternative;  // not defined
    template<size_t I, class T>
        struct variant_alternative<I, const T>;
    template<size_t I, class T>
        using variant_alternative_t = typename variant_alternative<I, T>::type;

    template<size_t I, class... Types>
        struct variant_alternative<I, variant<Types...>>;

    inline constexpr size_t variant_npos = -1;

    // 20.7.5, value access
    template<class T, class... Types>
        constexpr bool holds_alternative(const variant<Types...>&) noexcept;
    template<size_t I, class T>
        struct variant_alternative_t<I, variant<Types...>>& get(variant<Types...>&);
    template<size_t I, class... Types>
        constexpr variant_alternative_t<I, variant<Types...>>& get(variant<Types...>&&);
    template<size_t I, class... Types>
        constexpr const variant_alternative_t<I, variant<Types...>>& get(const variant<Types...>&);
    template<size_t I, class... Types>
        constexpr const variant_alternative_t<I, variant<Types...>>& get(const variant<Types...>&&);

§ 20.7.2 612
template<class T, class... Types>
constexpr T& get(variant<Types...>&);
template<class T, class... Types>
constexpr T&& get(variant<Types...>&&);
template<class T, class... Types>
constexpr const T& get(const variant<Types...>&);
template<class T, class... Types>
constexpr const T&& get(const variant<Types...>&&);

template<size_t I, class... Types>
constexpr add_pointer_t<variant_alternative_t<I, variant<Types...>>>
get_if(variant<Types...>*) noexcept;
template<size_t I, class... Types>
constexpr add_pointer_t<const variant_alternative_t<I, variant<Types...>>>
get_if(const variant<Types...>*) noexcept;

template<class T, class... Types>
constexpr add_pointer_t<T>
get_if(variant<Types...>*) noexcept;
template<class T, class... Types>
constexpr add_pointer_t<const T>
get_if(const variant<Types...>*) noexcept;

// 20.7.6, relational operators
template<class... Types>
constexpr bool operator==(const variant<Types...>&, const variant<Types...>&);
template<class... Types>
constexpr bool operator!=(const variant<Types...>&, const variant<Types...>&);
template<class... Types>
constexpr bool operator<(const variant<Types...>&, const variant<Types...>&);
template<class... Types>
constexpr bool operator>(const variant<Types...>&, const variant<Types...>&);
template<class... Types>
constexpr bool operator<=(const variant<Types...>&, const variant<Types...>&);
template<class... Types>
constexpr bool operator>=(const variant<Types...>&, const variant<Types...>&);

// 20.7.7, visitation
template<class Visitor, class... Variants>
constexpr see below visit(Visitor&&, Variants&&...);

// 20.7.8, class monostate
struct monostate;

// 20.7.9, monostate relational operators
constexpr bool operator==(monostate, monostate) noexcept;
constexpr strong_ordering operator<=>(monostate, monostate) noexcept; // 20.7.10, specialized algorithms

// 20.7.10, specialized algorithms
template<class... Types>
void swap(variant<Types...>&, variant<Types...>&) noexcept(see below);

// 20.7.11, class bad_variant_access
class bad_variant_access;

// 20.7.12, hash support
template<class T> struct hash;
template<class... Types> struct hash<variant<Types...>>;
template<> struct hash<monostate>;

20.7.3 Class template variant [variant.variant]

20.7.3.1 General [variant.variant.general]

namespace std {
    template<class... Types>
    class variant {
    public:

        // 20.7.3.2, constructors
        constexpr variant() noexcept;
        constexpr variant(const variant&);
        constexpr variant(variant&&) noexcept;
        template<class T>
        constexpr variant(T&&) noexcept;
        template<class T, class... Args>
        constexpr explicit variant(in_place_type_t<T>, Args&&...);
        template<class T, class U, class... Args>
        constexpr explicit variant(in_place_type_t<T>, initializer_list<U>, Args&&...);
        template<size_t I, class... Args>
        constexpr explicit variant(in_place_index_t<I>, Args&&...);
        template<size_t I, class U, class... Args>
        constexpr explicit variant(in_place_index_t<I>, initializer_list<U>, Args&&...);

        // 20.7.3.3, destructor
        ~variant();

        // 20.7.3.4, assignment
        constexpr variant& operator=(const variant&);
        constexpr variant& operator=(variant&&) noexcept;
        template<class T>
        variant& operator=(T&&) noexcept;

        // 20.7.3.5, modifiers
        template<class T, class... Args>
        T& emplace(Args&&...);
        template<class T, class U, class... Args>
        T& emplace(initializer_list<U>, Args&&...);
        template<size_t I, class... Args>
        variant_alternative_t<I, variant<Types...>>& emplace(Args&&...);
        template<size_t I, class U, class... Args>
        variant_alternative_t<I, variant<Types...>>& emplace(initializer_list<U>, Args&&...);

        // 20.7.3.6, value status
        constexpr bool valueless_by_exception() const noexcept;
        constexpr size_t index() const noexcept;

        // 20.7.3.7, swap
        void swap(variant&) noexcept;
    };
}

1 Any instance of variant at any given time either holds a value of one of its alternative types or holds no value. When an instance of variant holds a value of alternative type T, it means that a value of type T, referred to as the variant object’s contained value, is allocated within the storage of the variant object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate the contained value. The contained value shall be allocated in a region of the variant storage suitably aligned for all types in Types.

2 All types in Types shall meet the Cpp17Destructible requirements (Table 32).
A program that instantiates the definition of `variant` with no template arguments is ill-formed.

### Constructors

In the descriptions that follow, let \( i \) be in the range \([0, \text{sizeof...(Types)})\), and \( T_i \) be the \( i^\text{th} \) type in \( \text{Types} \).

```cpp
constexpr variant() noexcept;
```

**Constraints:** `is_default_constructible_v<T_0>` is true.

**Effects:** Constructs a `variant` holding a value-initialized value of type \( T_0 \).

**Postconditions:** `valueless_by_exception()` is false and `index()` is 0.

**Throws:** Any exception thrown by the value-initialization of \( T_0 \).

**Remarks:** This function is `constexpr` if and only if the value-initialization of the alternative type \( T_0 \) would satisfy the requirements for a `constexpr` function. The expression inside `noexcept` is equivalent to `is_nothrow_default_constructible_v<T_0>`.

[Note 1: See also class `monostate` — end note]

```cpp
constexpr variant(const variant& w);
```

**Effects:** If \( w \) holds a value, initializes the `variant` to hold the same alternative as \( w \) and direct-initializes the contained value with `get<j>(w)` where \( j \) is \( w.index() \). Otherwise, initializes the `variant` to not hold a value.

**Throws:** Any exception thrown by direct-initializing any \( T_i \) for all \( i \).

**Remarks:** This constructor is defined as deleted unless `is_copy_constructible_v<T_i>` is true for all \( i \). If `is_trivially_copy_constructible_v<T_i>` is true for all \( i \), this constructor is trivial.

```cpp
constexpr variant(variant&& w) noexcept;
```

**Constraints:** `is_move_constructible_v<T_i>` is true for all \( i \).

**Effects:** If \( w \) holds a value, initializes the `variant` to hold the same alternative as \( w \) and direct-initializes the contained value with `get<j>(std::move(w))`, where \( j \) is `w.index()`.

**Throws:** Any exception thrown by move-constructing any \( T_i \) for all \( i \).

**Remarks:** The expression inside `noexcept` is equivalent to the logical AND of `is_nothrow_move_constructible_v<T_i>` for all \( i \). If `is_trivially_move_constructible_v<T_i>` is true for all \( i \), this constructor is trivial.

```cpp
template<class T> constexpr variant(T&& t) noexcept;
```

Let \( T_j \) be a type that is determined as follows: build an imaginary function `FUN(T_i)` for each alternative type \( T_i \) for which \( T_i \cdot X = \{\text{std::forward<T>(t)}\} \) is well-formed for some invented variable \( x \). The overload `FUN(T_j)` selected by overload resolution for the expression `FUN(std::forward<T>(t))` defines the alternative \( T_j \) which is the type of the contained value after construction.

**Constraints:**

1. `sizeof...(Types)` is nonzero,
2. `is_same_v<remove_cvref_t<T>, variant>` is false,
3. `remove_cvref_t<T>` is neither a specialization of `in_place_type_t` nor a specialization of `in_place_index_t`,
4. `is_constructible_v<T_j, T>` is true, and
5. the expression `FUN(std::forward<T>(t))` (with `FUN` being the above-mentioned set of imaginary functions) is well-formed.

[Note 2: `variant<string, string> v("abc")` is ill-formed, as both alternative types have an equally viable constructor for the argument. — end note]
Effects: Initializes *this to hold the alternative type \( T_j \) and direct-initializes the contained value as if direct-non-list-initializing it with `std::forward<T>(t)`.

Postconditions: `holds_alternative<T>(*this)` is true.

Throws: Any exception thrown by the initialization of the selected alternative \( T_j \).

Remarks: The expression inside `noexcept` is equivalent to `is_nothrow_constructible_v<T_j, T>`. If \( T_j \)'s selected constructor is a constexpr constructor, this constructor is a constexpr constructor.

template<class T, class... Args> constexpr explicit variant(in_place_type_t<T>, Args&&... args);

Constraints:

(20.1) There is exactly one occurrence of \( T \) in `Types`...
(20.2) `is_constructible_v<T, Args...>` is true.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type \( T \) with the arguments `std::forward<Args>(args)`....

Postconditions: `holds_alternative<T>(*this)` is true.

Throws: Any exception thrown by calling the selected constructor of \( T \).

Remarks: If \( T \)'s selected constructor is a constexpr constructor, this constructor is a constexpr constructor.

template<class T, class U, class... Args>
constexpr explicit variant(in_place_type_t<T>, initializer_list<U> il, Args&&... args);

Constraints:

(25.1) There is exactly one occurrence of \( T \) in `Types`...
(25.2) `is_constructible_v<T, initializer_list<U>&, Args...>` is true.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type \( T \) with the arguments `il, std::forward<Args>(args)`....

Postconditions: `holds_alternative<T>(*this)` is true.

Throws: Any exception thrown by calling the selected constructor of \( T \).

Remarks: If \( T \)'s selected constructor is a constexpr constructor, this constructor is a constexpr constructor.

template<
size_t I, class... Args>
constexpr explicit variant(in_place_index_t<I>, Args&&... args);

Constraints:

(30.1) \( I \) is less than `sizeof...(Types)` and
(30.2) `is_constructible_v<T_I, Args...>` is true.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type \( T_I \) with the arguments `std::forward<Args>(args)`....

Postconditions: `index()` is \( I \).

Throws: Any exception thrown by calling the selected constructor of \( T_I \).

Remarks: If \( T_I \)'s selected constructor is a constexpr constructor, this constructor is a constexpr constructor.

template<
size_t I, class U, class... Args>
constexpr explicit variant(in_place_index_t<I>, initializer_list<U> il, Args&&... args);

Constraints:

(35.1) \( I \) is less than `sizeof...(Types)` and
(35.2) `is_constructible_v<T_I, initializer_list<U>&, Args...>` is true.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type \( T_I \) with the arguments `il, std::forward<Args>(args)`....

Postconditions: `index()` is \( I \).
Remarks: If \( T_j \)'s selected constructor is a constexpr constructor, this constructor is a constexpr constructor.

20.7.3.3 Destructor [variant.dtor]

\[ \text{~\texttt{variant();}} \]

Effects: If \texttt{valueless\_by\_exception()} is \texttt{false}, destroys the currently contained value.

Remarks: If \texttt{is\_trivially\_destructible\_v<T_i>} is \texttt{true} for all \( T_i \), then this destructor is trivial.

20.7.3.4 Assignment [variant.assign]

\[ \texttt{constexpr variant\& operator=}=(\text{\texttt{const variant\&}} \text{\texttt{rhs});}} \]

Let \( j \) be \texttt{rhs.index()}. 

Effects:

(2.1) If neither \*\texttt{this} nor \texttt{rhs} holds a value, there is no effect.

(2.2) Otherwise, if \*\texttt{this} holds a value but \texttt{rhs} does not, destroys the value contained in \*\texttt{this} and sets \*\texttt{this} to not hold a value.

(2.3) Otherwise, if \texttt{index()} == \( j \), assigns the value contained in \texttt{rhs} to the value contained in \*\texttt{this}.

(2.4) Otherwise, if either \texttt{is\_nothrow\_copy\_constructible\_v<T_j>} is \texttt{true} or \texttt{is\_nothrow\_move\_constructible\_v<T_j>} is \texttt{false}, equivalent to \texttt{emplace<}(\texttt{get<}(\texttt{rhs})).

(2.5) Otherwise, equivalent to \texttt{operator=}=(\texttt{variant(rhs)}).

Postconditions: \texttt{index()} == \texttt{rhs.index()}. 

Returns: \*\texttt{this}.

Remarks: This operator is defined as deleted unless \texttt{is\_copy\_constructible\_v<T_i>} && \texttt{is\_copy\_assignable\_v<T_i>} is \texttt{true} for all \( i \). If \texttt{is\_trivially\_copy\_constructible\_v<T_i>} && \texttt{is\_trivially\_copy\_assignable\_v<T_i>} && \texttt{is\_trivially\_destructible\_v<T_i>} is \texttt{true} for all \( i \), this assignment operator is trivial.

\[ \text{\texttt{constexpr variant\& operator=}=(\text{\texttt{variant\&}} \text{\texttt{rhs);}} \text{\texttt{ noexcept(see below);}}} \]

Let \( j \) be \texttt{rhs.index()}. 

Constraints: \texttt{is\_move\_constructible\_v<T_i>} && \texttt{is\_move\_assignable\_v<T_i>} is \texttt{true} for all \( i \).

Effects:

(8.1) If neither \*\texttt{this} nor \texttt{rhs} holds a value, there is no effect.

(8.2) Otherwise, if \*\texttt{this} holds a value but \texttt{rhs} does not, destroys the value contained in \*\texttt{this} and sets \*\texttt{this} to not hold a value.

(8.3) Otherwise, if \texttt{index()} == \( j \), assigns \texttt{get<}(\texttt{std::move(rhs)}) to the value contained in \*\texttt{this}.

(8.4) Otherwise, equivalent to \texttt{emplace<}(\texttt{get<}(\texttt{std::move(rhs)})).

Returns: \*\texttt{this}.

Remarks: If \texttt{is\_trivially\_move\_constructible\_v<T_i>} && \texttt{is\_trivially\_move\_assignable\_v<T_i>} && \texttt{is\_trivially\_destructible\_v<T_i>} is \texttt{true} for all \( i \), this assignment operator is trivial. The expression inside \texttt{noexcept} is equivalent to \texttt{is\_nothrow\_move\_constructible\_v<T_i>} && \texttt{is\_nothrow\_move\_assignable\_v<T_i>} for all \( i \).

(10.1) If an exception is thrown during the call to \( T_j \)'s move construction (with \( j \) being \texttt{rhs.index()}), the \texttt{variant} will hold no value.

(10.2) If an exception is thrown during the call to \( T_j \)'s move assignment, the state of the contained value is as defined by the exception safety guarantee of \( T_j \)'s move assignment; \texttt{index()} will be \( j \).

\[ \text{\texttt{template<\texttt{class T}\> variant\& operator=}=(T\&\& t) \text{\texttt{ noexcept(see below);}} \] 

Let \( T_j \) be a type that is determined as follows: build an imaginary function \texttt{FUN(T_j)} for each alternative \texttt{type T_i} for which \( T_i \ x[] = \{\texttt{std::forward<T>(t)}\} \) is well-formed for some invented variable \( x \). The overload \texttt{FUN(T_j)} selected by overload resolution for the expression \texttt{FUN(std::forward<T>(t))} defines the alternative \( T_j \) which is the type of the contained value after assignment.
Constraints:

- \(\text{is\_same\_v<remove\_cvref\_t<T>, variant}> \) is false,
- \(\text{is\_assignable\_v<T, T> \\ & \& \text{is\_constructible\_v<T_j, T}> \) is true, and
- the expression \(\text{FUN}(\text{std::forward<T>(t)})\) (with \(\text{FUN}\) being the above-mentioned set of imaginary functions) is well-formed.

[Note 1:

\[
\text{variant<string, string> v; v = "abc";}
\]

is ill-formed, as both alternative types have an equally viable constructor for the argument. — end note]

Effects:

- If \(*\text{this}\) holds a \(T_j\), assigns \(\text{std::forward<T>(t)}\) to the value contained in \(*\text{this}\).
- Otherwise, if \(\text{is\_nothrow\_constructible\_v<T_j, T> || !\text{is\_nothrow\_move\_constructible\_v<T_j, T>}}\) is true, equivalent to \(\text{emplace\_v<T}\)\(>(\text{std::forward<T>(t)})\).
- Otherwise, equivalent to \(\text{operator=(variant(\text{std::forward<T>(t)})\).}\)

Postconditions: \(\text{holds\_alternative\_v<T_j>(*\text{this})}\) is true, with \(T_j\) selected by the imaginary function overload resolution described above.

Returns: \(*\text{this}\).

Remarks: The expression inside \text{noexcept} is equivalent to:

\[
\text{is\_nothrow\_assignable::<T, Args...> \\ & \& \text{is\_nothrow\_constructible::<T, Args...>}
\]

- If an exception is thrown during the assignment of \(\text{std::forward<T>(t)}\) to the value contained in \(*\text{this}\), the state of the contained value and \(t\) are as defined by the exception safety guarantee of the assignment expression; \text{valueless\_by\_exception()} will be \text{false}.
- If an exception is thrown during the initialization of the contained value, the \text{variant} object is permitted to not hold a value.

20.7.3.5 Modifiers

\text{template<class T, class... Args> T& emplace(Args&&... args);}  

Constraints: \(\text{is\_constructible\_v<T, Args...> \) is true, and \(T\) occurs exactly once in \(\text{Types}\).

Effects: Equivalent to:

\[
\text{return emplace\_/\_/\text{I}(\text{std::forward<Args>(args)...});}
\]

where \(I\) is the zero-based index of \(T\) in \(\text{Types}\).

\text{template<class T, class U, class... Args> T& emplace(initializer_list<U> il, Args&&... args);}  

Constraints: \(\text{is\_constructible\_v<T, initializer\_list<U>&, Args...> \) is true, and \(T\) occurs exactly once in \(\text{Types}\).

Effects: Equivalent to:

\[
\text{return emplace\_/\_/\text{I}(il, \text{std::forward<Args>(args)...});}
\]

where \(I\) is the zero-based index of \(T\) in \(\text{Types}\).

\text{template<size_t I, class... Args>}  

\[
\text{variant\_alternative\_t\<I, variant<\text{Types}...>\>& emplace(Args&&... args);}  
\]

Mandates: \(I < \text{sizeof...(\text{Types})}\).

Constraints: \(\text{is\_constructible\_v<T_I, Args...> \) is true.

Effects: Destroys the currently contained value if \text{valueless\_by\_exception()} is \text{false}. Then initializes the contained value as if direct-non-list-initializing a value of type \(T_I\) with the arguments \(\text{std::forward<Args>(args)...}\).

Postconditions: \text{index()} is \(I\).

Returns: A reference to the new contained value.

Throws: Any exception thrown during the initialization of the contained value.
Remarks: If an exception is thrown during the initialization of the contained value, the variant is permitted to not hold a value.

```cpp
template<size_t I, class U, class... Args>
variant_alternative_t<I, variant<Types...>>& emplace(initializer_list<U> il, Args&&... args);
```

Mandates: $I < \text{sizeof...}(\text{Types})$.

Constraints: $\text{is_constructible_v}_I(\text{initializer_list}<\text{U}>, \text{Args}...)$ is true.

Effects: Destroys the currently contained value if valueless_by_exception() is false. Then initializes the contained value as if direct-non-list-initializing a value of type $T_I$ with the arguments $\text{il}$, std::forward<Args>(args)... .

Postconditions: index() is $I$.

Returns: A reference to the new contained value.

Throws: Any exception thrown during the initialization of the contained value.

Remarks: If an exception is thrown during the initialization of the contained value, the variant is permitted to not hold a value.

### 20.7.3.6 Value status

```cpp
constexpr bool valueless_by_exception() const noexcept;
```

Effects: Returns false if and only if the variant holds a value.

[Note 1: A variant might not hold a value if an exception is thrown during a type-changing assignment or emplacement. The latter means that even a variant<float, int> can become valueless_by_exception(), for instance by

```cpp
struct S { operator int() { throw 42; } }; variant<float, int> v(12.f); v.emplace<1>(S());
```

— end note]

```cpp
constexpr size_t index() const noexcept;
```

Effects: If valueless_by_exception() is true, returns variant_npos. Otherwise, returns the zero-based index of the alternative of the contained value.

### 20.7.3.7 Swap

```cpp
void swap(variant& rhs) noexcept(see below);
```

Mandates: is_move_constructible_v$_I(\text{for all } I)$.

Preconditions: Lvalues of type $T_I$ are swappable (16.4.4.3).

Effects:

1. If valueless_by_exception() && rhs.valueless_by_exception() no effect.
2. Otherwise, if index() == rhs.index(), calls swap(get<i>(*this), get<i>(rhs)) where $i$ is index().
3. Otherwise, exchanges values of rhs and *this.

Throws: If index() == rhs.index(), any exception thrown by swap(get<i>(*this), get<i>(rhs)) with $i$ being index(). Otherwise, any exception thrown by the move constructor of $T_i$ or $T_j$ with $i$ being index() and $j$ being rhs.index().

Remarks: If an exception is thrown during the call to function swap(get<i>(*this), get<i>(rhs)), the states of the contained values of *this and of rhs are determined by the exception safety guarantee of swap for lvalues of $T_i$ with $i$ being index(). If an exception is thrown during the exchange of the values of *this and rhs, the states of the values of *this and of rhs are determined by the exception safety guarantee of variant's move constructor. The expression inside noexcept is equivalent to the logical AND of is_nothrow_move_constructible_v$_I(\text{for all } I)$.
20.7.4 variant helper classes

```
#define variant_helper
```

```
template<class T> struct variant_size;
```

All specializations of `variant_size` meet the `Cpp17UnaryTypeTrait` requirements (20.15.2) with a base characteristic of `integral_constant<size_t, N>` for some `N`.

```
template<class T> struct variant_size<const T>;
```

Let `VS` denote `variant_size<T>` of the cv-unqualified type `T`. Then each specialization of the template meets the `Cpp17UnaryTypeTrait` requirements (20.15.2) with a base characteristic of `integral_constant<size_t, VS::value>`.

```
template<class... Types>
struct variant_size<variant<Types...>> : integral_constant<size_t, sizeof...(Types)> { };
```

```
template<size_t I, class T> struct variant_alternative<I, const T>;
```

Let `VA` denote `variant_alternative<I, T>` of the cv-unqualified type `T`. Then each specialization of the template meets the `Cpp17TransformationTrait` requirements (20.15.2) with a member typedef `type` that names the type `add_const_t<VA::type>`.

```
variant_alternative<I, variant<Types...>>::type
```

**Mandates:** `I < sizeof...(Types)`.

**Type:** The type `T_I`.

20.7.5 Value access

```
template<class T, class... Types>
constexpr bool holds_alternative(const variant<Types...>& v) noexcept;
```

**Mandates:** The type `T` occurs exactly once in `Types`.

**Returns:** `true` if `index()` is equal to the zero-based index of `T` in `Types`.

```
template<size_t I, class T, class... Types>
constexpr variant_alternative_t<I, variant<Types...>>& get(variant<Types...>& v);
template<size_t I, class T, class... Types>
constexpr variant_alternative_t<I, variant<Types...>>&& get(variant<Types...>&& v);
template<size_t I, class T, class... Types>
constexpr const variant_alternative_t<I, variant<Types...>>& get(const variant<Types...>& v);
template<size_t I, class T, class... Types>
constexpr const variant_alternative_t<I, variant<Types...>>&& get(const variant<Types...>&& v);
```

**Mandates:** `I < sizeof...(Types)`.

**Effects:** If `v.index()` is `I`, returns a reference to the object stored in the `variant`. Otherwise, throws an exception of type `bad_variant_access`.

```
template<class T, class... Types>
constexpr T& get(variant<Types...>& v);
template<class T, class... Types>
constexpr T&& get(variant<Types...>&& v);
template<class T, class... Types>
constexpr const T& get(const variant<Types...>& v);
template<class T, class... Types>
constexpr const T&& get(const variant<Types...>&& v);
```

**Mandates:** The type `T` occurs exactly once in `Types`.

**Effects:** If `v` holds a value of type `T`, returns a reference to that value. Otherwise, throws an exception of type `bad_variant_access`.

```
template<size_t I, class T, class... Types>
constexpr add_pointer_t<variant_alternative_t<I, variant<Types...>>> get_if(variant<Types...>& v) noexcept;
template<size_t I, class T, class... Types>
constexpr add_pointer_t<const variant_alternative_t<I, variant<Types...>>> get_if(const variant<Types...>& v) noexcept;
```

**Mandates:** `I < sizeof...(Types)`.

**Returns:** A pointer to the value stored in the `variant`, if `v != nullptr` and `v->index()` == `I`. Otherwise, returns `nullptr`.

§ 20.7.5 620
template<class T, class... Types>
constexpr add_pointer_t<T>
get_if(variant<Types...>* v) noexcept;

template<class T, class... Types>
constexpr add_pointer_t<const T>
get_if(const variant<Types...>* v) noexcept;

Mandates: The type T occurs exactly once in Types.

Effects: Equivalent to: return get_if<i>(v); with i being the zero-based index of T in Types.

20.7.6 Relational operators

template<class... Types>
constexpr bool operator==(const variant<Types...>& v, const variant<Types...>& w);

Mandates: get<i>(v) == get<i>(w) is a valid expression that is convertible to bool, for all i.

Returns: If v.index() != w.index(), false; otherwise if v.valueless_by_exception(), true; otherwise get<i>(v) == get<i>(w) with i being v.index().

template<class... Types>
constexpr bool operator!=(const variant<Types...>& v, const variant<Types...>& w);

Mandates: get<i>(v) != get<i>(w) is a valid expression that is convertible to bool, for all i.

Returns: If v.index() != w.index(), true; otherwise if v.valueless_by_exception(), false; otherwise get<i>(v) != get<i>(w) with i being v.index().

template<class... Types>
constexpr bool operator<(const variant<Types...>& v, const variant<Types...>& w);

Mandates: get<i>(v) < get<i>(w) is a valid expression that is convertible to bool, for all i.

Returns: If w.valueless_by_exception(), false; otherwise if v.valueless_by_exception(), true; otherwise, if v.index() < w.index(), true; otherwise if v.index() > w.index(), false; otherwise get<i>(v) < get<i>(w) with i being v.index().

template<class... Types>
constexpr bool operator>(const variant<Types...>& v, const variant<Types...>& w);

Mandates: get<i>(v) > get<i>(w) is a valid expression that is convertible to bool, for all i.

Returns: If v.valueless_by_exception(), false; otherwise if w.valueless_by_exception(), true; otherwise, if v.index() > w.index(), true; otherwise if v.index() < w.index(), false; otherwise get<i>(v) > get<i>(w) with i being v.index().

template<class... Types>
constexpr bool operator<=(const variant<Types...>& v, const variant<Types...>& w);

Mandates: get<i>(v) <= get<i>(w) is a valid expression that is convertible to bool, for all i.

Returns: If v.valueless_by_exception(), true; otherwise if w.valueless_by_exception(), false; otherwise, if v.index() < w.index(), true; otherwise if v.index() > w.index(), false; otherwise get<i>(v) <= get<i>(w) with i being v.index().

template<class... Types>
constexpr bool operator>=(const variant<Types...>& v, const variant<Types...>& w);

Mandates: get<i>(v) >= get<i>(w) is a valid expression that is convertible to bool, for all i.

Returns: If w.valueless_by_exception(), true; otherwise if v.valueless_by_exception(), false; otherwise, if v.index() > w.index(), true; otherwise if v.index() < w.index(), false; otherwise get<i>(v) >= get<i>(w) with i being v.index().

template<class... Types> requires (three_way_comparable<Types> && ...)
constexpr common_comparison_category_t<compare_three_way_result_t<Types>...>
operator<=>(const variant<Types...>& v, const variant<Types...>& w);

Effects: Equivalent to:

if (v.valueless_by_exception() && w.valueless_by_exception())
    return strong_ordering::equal;

§ 20.7.6 621
if (v.valueless_by_exception()) return strong_ordering::less;
if (w.valueless_by_exception()) return strong_ordering::greater;
if (auto c = v.index() <=> w.index(); c != 0) return c;
return get<i>(v) <=> get<i>(w);

with i being v.index().

20.7.7 Visitation

```cpp
template<class Visitor, class... Variants>
constexpr see below visit(Visitor&& vis, Variants&&... vars);
template<class R, class Visitor, class... Variants>
constexpr R visit(Visitor&& vis, Variants&&... vars);
```

1 Let n be sizeof...(Variants). Let m be a pack of n values of type size_t. Such a pack is called valid if 0 ≤ m_i < variant_size_v<remove_reference_t<Variants_i>> for all 0 ≤ i < n. For each valid pack m, let e(m) denote the expression:

```
INVoke(std::forward<Visitor>(vis), get<m>(std::forward<Variants>(vars))...) // see 20.14.4
```

for the first form and

```
INVoke<R>(std::forward<Visitor>(vis), get<m>(std::forward<Variants>(vars))...) // see 20.14.4
```

for the second form.

2 Mandates: For each valid pack m, e(m) is a valid expression. All such expressions are of the same type and value category.

3 Returns: e(m), where m is the pack for which m_i is vars_i.index() for all 0 ≤ i < n. The return type is decltype(e(m)) for the first form.

4 Throws: bad_variant_access if any variant in vars is valueless_by_exception().

5 Complexity: For n ≤ 1, the invocation of the callable object is implemented in constant time, i.e., for n = 1, it does not depend on the number of alternative types of Variants_i. For n > 1, the invocation of the callable object has no complexity requirements.

20.7.8 Class monostate

```cpp
struct monostate{}
```

1 The class monostate can serve as a first alternative type for a variant to make the variant type default constructible.

20.7.9 monostate relational operators

```cpp
constexpr bool operator==(monostate, monostate) noexcept { return true; }
constexpr strong_ordering operator<=>(monostate, monostate) noexcept
{
    return strong_ordering::equal;
}
```

1 [Note 1: monostate objects have only a single state; they thus always compare equal. — end note]

20.7.10 Specialized algorithms

```cpp
template<class... Types>
void swap(variant<Types...>& v, variant<Types...>& w) noexcept(see below);
```

1 Constraints: is_move_constructible_v<T_i> && is_swappable_v<T_i> is true for all i.

2 Effects: Equivalent to v.swap(w).

3 Remarks: The expression inside noexcept is equivalent to noexcept(v.swap(w)).

20.7.11 Class bad_variant_access

```cpp
class bad_variant_access : public exception {
    public:
        // see 17.9.3 for the specification of the special member functions
        const char* what() const noexcept override;
};
```
Objects of type `bad_variant_access` are thrown to report invalid accesses to the value of a `variant` object.

```cpp
const char* what() const noexcept override;
```

Returns: An implementation-defined Ntbs.

### 20.7.12 Hash support

```cpp
template<class... Types> struct hash<variant<Types...>>;
```

The specialization `hash<variant<Types...>>` is enabled (20.14.19) if and only if every specialization in `hash<remove_const_t<Types>>...` is enabled. The member functions are not guaranteed to be noexcept.

```cpp
template<> struct hash<monostate>;
```

The specialization is enabled (20.14.19).

### 20.8 Storage for any type

#### 20.8.1 General

Subclause 20.8 describes components that C++ programs may use to perform operations on objects of a discriminated type.

**Note 1:** The discriminated type can contain values of different types but does not attempt conversion between them, i.e., 5 is held strictly as an `int` and is not implicitly convertible either to "5" or to 5.0. This indifference to interpretation but awareness of type effectively allows safe, generic containers of single values, with no scope for surprises from ambiguous conversions. — end note

#### 20.8.2 Header `<any>` synopsis

```cpp
namespace std {
  // 20.8.3, class bad_any_cast
  class bad_any_cast;

  // 20.8.4, class any
  class any;

  // 20.8.5, non-member functions
  void swap(any& x, any& y) noexcept;
  template<class T, class... Args>
  any make_any(Args&&... args);
  template<class T, class U, class... Args>
  any make_any(initializer_list<U> il, Args&&... args);

  template<class T>
  T any_cast(const any& operand);
  template<class T>
  T any_cast(any& operand);
  template<class T>
  T any_cast(any&& operand);

  template<class T>
  const T* any_cast(const any* operand) noexcept;
  template<class T>
  T* any_cast(any* operand) noexcept;
}
```

### 20.8.3 Class bad_any_cast

```cpp
class bad_any_cast : public bad_cast {
  public:
    // see 17.9.3 for the specification of the special member functions
    const char* what() const noexcept override;
};
```
Objects of type `bad_any_cast` are thrown by a failed `any_cast` (20.8.5).

```
const char* what() const noexcept override;
```

Returns: An implementation-defined NTBS.

### 20.8.4 Class `any`

#### 20.8.4.1 General

```
namespace std {
    class any {
        public:
            // 20.8.4.2, construction and destruction
            constexpr any() noexcept;
            any(const any& other);
            any(any&& other) noexcept;
            template<class T>
                any(T&& value);
            template<class T, class... Args>
                explicit any(in_place_type_t<T>, Args&&...);
            template<class T, class U, class... Args>
                explicit any(in_place_type_t<T>, initializer_list<U>, Args&&...);
            ~any();

            // 20.8.4.3, assignments
            any& operator=(const any& rhs);
            any& operator=(any&& rhs) noexcept;
            template<class T>
                any& operator=(T&& rhs);

            // 20.8.4.4, modifiers
            template<class T, class... Args>
                decay_t<T>& emplace(Args&&...);
            template<class T, class U, class... Args>
                decay_t<T>& emplace(initializer_list<U>, Args&&...);
            void reset() noexcept;
            void swap(any& rhs) noexcept;

            // 20.8.4.5, observers
            bool has_value() const noexcept;
            const type_info& type() const noexcept;
    };
}
```

1. An object of class `any` stores an instance of any type that meets the constructor requirements or it has no value, and this is referred to as the *state* of the class `any` object. The stored instance is called the *contained value*. Two states are equivalent if either they both have no value, or they both have a value and the contained values are equivalent.

2. The non-member `any_cast` functions provide type-safe access to the contained value.

3. Implementations should avoid the use of dynamically allocated memory for a small contained value. However, any such small-object optimization shall only be applied to types `T` for which `is_nothrow_move_constructible_v<T>` is true.

   [Example 1: A contained value of type `int` could be stored in an internal buffer, not in separately-allocated memory. — end example]

#### 20.8.4.2 Construction and destruction

```
constexpr any() noexcept;
```

1. *Postconditions: has_value() is false.*
any(const any& other);

Effects: If other.has_value() is false, constructs an object that has no value. Otherwise, equivalent to any(in_place_type<T>, any_cast< const T& >(other)) where T is the type of the contained value.

Throws: Any exceptions arising from calling the selected constructor for the contained value.

any(any&& other) noexcept;

Effects: If other.has_value() is false, constructs an object that has no value. Otherwise, constructs an object of type any that contains either the contained value of other, or contains an object of the same type constructed from the contained value of other considering that contained value as an rvalue.

template<class T>
any(T&& value);

Let VT be decay_t<T>.

Constraints: VT is not the same type as any, VT is not a specialization of in_place_type_t, and is_copy_constructible_v<VT> is true.

Preconditions: VT meets the Cpp17CopyConstructible requirements.

Effects: Constructs an object of type any that contains an object of type VT direct-initialized with std::forward<T>(value).

Throws: Any exception thrown by the selected constructor of VT.

template<class T, class... Args>
explicit any(in_place_type_t<T>, Args&&... args);

Let VT be decay_t<T>.

Constraints: is_copy_constructible_v<VT> is true and is_constructible_v<VT, Args...> is true.

Preconditions: VT meets the Cpp17CopyConstructible requirements.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type VT with the arguments std::forward<Args>(args)....

Postconditions: *this contains a value of type VT.

Throws: Any exception thrown by the selected constructor of VT.

template<class T, class U, class... Args>
explicit any(in_place_type_t<T>, initializer_list<U> il, Args&&... args);

Let VT be decay_t<T>.

Constraints: is_copy_constructible_v<VT> is true and is_constructible_v<VT, initializer_list<U>&, Args...> is true.

Preconditions: VT meets the Cpp17CopyConstructible requirements.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type VT with the arguments il, std::forward<Args>(args)....

Postconditions: *this contains a value.

Throws: Any exception thrown by the selected constructor of VT.

~any();

Effects: As if by reset().

20.8.4.3 Assignment

any& operator=(const any& rhs);

Effects: As if by any(rhs).swap(*this). No effects if an exception is thrown.

Returns: *this.

Throws: Any exceptions arising from the copy constructor for the contained value.
any& operator=(any&& rhs) noexcept;

Effects: As if by any(std::move(rhs)).swap(*this).

Postconditions: The state of *this is equivalent to the original state of rhs.

Returns: *this.

template<class T>
any& operator=(T&& rhs);

Let VT be decay_t<T>.

Constraints: VT is not the same type as any and is_copy_constructible_v<VT> is true.

Preconditions: VT meets the Cpp17CopyConstructible requirements.

Effects: Constructs an object tmp of type any that contains an object of type VT direct-initialized with std::forward<T>(rhs), and tmp.swap(*this). No effects if an exception is thrown.

Returns: *this.

Throws: Any exception thrown by the selected constructor of VT.

20.8.4.4 Modifiers

[any.modifiers]

template<class T, class... Args>
decay_t<T>& emplace(Args&&... args);

Let VT be decay_t<T>.

Constraints: is_copy_constructible_v<VT> is true and is_constructible_v<VT, Args...> is true.

Preconditions: VT meets the Cpp17CopyConstructible requirements.

Effects: Calls reset(). Then initializes the contained value as if direct-non-list-initializing an object of type VT with the arguments std::forward<Args>(args)....

Postconditions: *this contains a value.

Returns: A reference to the new contained value.

Throws: Any exception thrown by the selected constructor of VT.

Remarks: If an exception is thrown during the call to VT’s constructor, *this does not contain a value, and any previously contained value has been destroyed.

template<class T, class U, class... Args>
decay_t<T>& emplace(initializer_list<U> il, Args&&... args);

Let VT be decay_t<T>.

Constraints: is_copy_constructible_v<VT> is true and is_constructible_v<VT, initializer_list<U>, Args...> is true.

Preconditions: VT meets the Cpp17CopyConstructible requirements.

Effects: Calls reset(). Then initializes the contained value as if direct-non-list-initializing an object of type VT with the arguments il, std::forward<Args>(args)....

Postconditions: *this contains a value.

Returns: A reference to the new contained value.

Throws: Any exception thrown by the selected constructor of VT.

Remarks: If an exception is thrown during the call to VT’s constructor, *this does not contain a value, and any previously contained value has been destroyed.

void reset() noexcept;

Effects: If has_value() is true, destroys the contained value.

Postconditions: has_value() is false.

void swap(any& rhs) noexcept;

Effects: Exchanges the states of *this and rhs.
20.8.4.5 Observers

bool has_value() const noexcept;

Returns: true if *this contains an object, otherwise false.

const type_info& type() const noexcept;

Returns: typeid(T) if *this has a contained value of type T, otherwise typeid(void).

[Note 1: Useful for querying against types known either at compile time or only at runtime. — end note]

20.8.5 Non-member functions

void swap(any& x, any& y) noexcept;

Effects: Equivalent to x.swap(y).

template<class T, class... Args>
any make_any(Args&&... args);

Effects: Equivalent to: return any(in_place_type<T>, std::forward<Args>(args)...);

template<class T, class U, class... Args>
any make_any(initializer_list<U> il, Args&&... args);

Effects: Equivalent to: return any(in_place_type<T>, il, std::forward<Args>(args)...);

template<class T>
T any_cast(const any& operand);
template<class T>
T any_cast(any& operand);
template<class T>
T any_cast(any&& operand);

Let U be the type remove_cvref_t<T>.

Mandates: For the first overload, is_constructible_v<T, const U&> is true. For the second overload, is_constructible_v<T, U&> is true. For the third overload, is_constructible_v<T, U> is true.

Returns: For the first and second overload, static_cast<T>(*any_cast<U>(&operand)). For the third overload, static_cast<T>((std::move(*any_cast<U>(&operand)))).

Throws: bad_any_cast if operand.type() != typeid(remove_reference_t<T>).

[Example 1:

any x(5);

assert(any_cast<int>(x) == 5); // cast to value
assert(any_cast<const int&>(x) == 10); // cast to reference
assert(any_cast<const int*>(x) == 10);

x = "Meow"; // x holds const char*
assert(strcmp(any_cast<const char*>(x), "Meow") == 0);
any_cast<const char*>(x) = "Harry";
assert(strcmp(any_cast<const char*>(x), "Harry") == 0);

x = string("Meow"); // x holds string
string s, s2("Jane");
s = move(any_cast<string&>(x)); // move from any
assert(s == "Meow");
any_cast<string&>(x) = move(s2); // move to any
assert(any_cast<const string&>(x) == "Jane");

string cat("Meow");
const any y(cat); // const y holds string
assert(any_cast<const string&>(y) == cat);

any_cast<string&>(y); // error: cannot any_cast away const
— end example]
template<class T>
const T* any_cast(const any* operand) noexcept;

T* any_cast(any* operand) noexcept;

9 Returns: If operand != nullptr && operand->type() == typeid(T), a pointer to the object contained by operand; otherwise, nullptr.

[Example 2:
  bool is_string(const any& operand) {
    return any_cast<string>(operand) != nullptr;
  }

  —end example]

20.9 Bitsets

20.9.1 Header <bitset> synopsis

The header <bitset> defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

```cpp
#include <string>
#include <iosfwd>  // for istream (29.7.1), ostream (29.7.2), see 29.3.1

namespace std {
  template<size_t N> class bitset;

  // 20.9.4, bitset operators
  template<size_t N>
  bitset<N> operator&(const bitset<N>&, const bitset<N>&) noexcept;
  template<size_t N>
  bitset<N> operator|(const bitset<N>&, const bitset<N>&) noexcept;
  template<size_t N>
  bitset<N> operator^(const bitset<N>&, const bitset<N>&) noexcept;
  template<class charT, class traits, size_t N>
  basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
  template<class charT, class traits, size_t N>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
}
```

20.9.2 Class template bitset

20.9.2.1 General

```cpp
namespace std {
  template<size_t N> class bitset {
  public:
    // bit reference
class reference {
      friend class bitset;
      reference() noexcept;

      public:
      reference(const reference&) = default;
      ~reference();
      reference& operator=(bool x) noexcept;  // for b[i] = x;
      reference& operator=(const reference&) noexcept;  // for b[i] = b[j];
      bool operator~() const noexcept;  // flips the bit
      operator bool() const noexcept;  // for x = b[i];
      reference& flip() noexcept;  // for b[i].flip();
    };

    // 20.9.2.2, constructors
    constexpr bitset() noexcept;
    constexpr bitset(unsigned long long val) noexcept;
```
template<class charT, class traits, class Allocator>
explicit bitset(
    const basic_string<charT, traits, Allocator>& str,
    typename basic_string<charT, traits, Allocator>::size_type pos = 0,
    typename basic_string<charT, traits, Allocator>::size_type n
    = basic_string<charT, traits, Allocator>::npos,
    charT zero = charT('0'),
    charT one = charT('1'))
    : bitset<charT, Traits, Alloc>(str.c_str(), pos, n, zero, one)
{
    // 20.9.2.3, bitset operations
    bitset<N>& operator&=(const bitset<N>& rhs) noexcept;
    bitset<N>& operator|=(const bitset<N>& rhs) noexcept;
    bitset<N>& operator^=(const bitset<N>& rhs) noexcept;
    bitset<N>& operator<<=(size_t pos) noexcept;
    bitset<N>& operator>>=(size_t pos) noexcept;
    bitset<N>& set() noexcept;
    bitset<N>& set(size_t pos, bool val = true);
    bitset<N>& reset() noexcept;
    bitset<N>& reset(size_t pos);
    bitset<N> operator~() const noexcept;
    bitset<N>& flip() noexcept;
    bitset<N>& flip(size_t pos);

    // element access
    constexpr bool operator[](size_t pos) const; // for b[i];
    reference operator[](size_t pos); // for b[i];

    unsigned long to_ulong() const;
    unsigned long long to_ullong() const;
    template<class charT = char,
    class traits = char_traits<charT>,
    class Allocator = allocator<charT>>
    basic_string<charT, traits, Allocator>
    to_string(charT zero = charT('0'), charT one = charT('1')) const;

    size_t count() const noexcept;
    constexpr size_t size() const noexcept;

    bool operator==(const bitset<N>& rhs) const noexcept;
    bool test(size_t pos) const;
    bool all() const noexcept;
    bool any() const noexcept;
    bool none() const noexcept;
    bitset<N> operator<<(size_t pos) const noexcept;
    bitset<N> operator>>(size_t pos) const noexcept;
};

// 20.9.3, hash support

1 The class template `bitset<N>` describes an object that can store a sequence consisting of a fixed number of
bits, N.

2 Each bit represents either the value zero (reset) or one (set). To **toggle** a bit is to change the value zero to
one, or the value one to zero. Each bit has a non-negative position pos. When converting between an object
of class `bitset<N>` and a value of some integral type, bit position pos corresponds to the **bit value** `1 << pos`.
The integral value corresponding to two or more bits is the sum of their bit values.

§ 20.9.2.1 629
The functions described in 20.9.2 can report three kinds of errors, each associated with a distinct exception:

- an invalid-argument error is associated with exceptions of type invalid_argument (19.2.5);
- an out-of-range error is associated with exceptions of type out_of_range (19.2.7);
- an overflow error is associated with exceptions of type overflow_error (19.2.10).

20.9.2.2 Constructors

```cpp
constexpr bitset() noexcept;
```

**Effects:** Initializes all bits in *this to zero.

```cpp
constexpr bitset(unsigned long long val) noexcept;
```

**Effects:** Initializes the first \( M \) bit positions to the corresponding bit values in \( \text{val} \). \( M \) is the smaller of \( N \) and the number of bits in the value representation (6.8) of unsigned long long. If \( M < N \), the remaining bit positions are initialized to zero.

```cpp
template<class charT, class traits, class Allocator>
explicit bitset(
    const basic_string<charT, traits, Allocator>& str,
    typename basic_string<charT, traits, Allocator>::size_type pos = 0,
    typename basic_string<charT, traits, Allocator>::size_type n = basic_string<charT, traits, Allocator>::npos,
    charT zero = charT('0'),
    charT one = charT('1'));
```

**Effects:** Determines the effective length \( rlen \) of the initializing string as the smaller of \( n \) and \( \text{str.size()} - \text{pos} \). Initializes the first \( M \) bit positions to values determined from the corresponding characters in the string \( \text{str} \). \( M \) is the smaller of \( N \) and \( rlen \).

An element of the constructed object has value zero if the corresponding character in \( \text{str} \), beginning at position \( \text{pos} \), is \( \text{zero} \). Otherwise, the element has the value one. Character position \( \text{pos} + M - 1 \) corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions.

If \( M < N \), remaining bit positions are initialized to zero.

The function uses \( \text{traits::eq} \) to compare the character values.

```cpp
template<class charT>
explicit bitset(
    const charT* str,
    typename basic_string<charT>::size_type n = basic_string<charT>::npos,
    charT zero = charT('0'),
    charT one = charT('1'));
```

**Effects:** As if by:

```cpp
bitset(n == basic_string<charT>::npos
    ? basic_string<charT>(str)
    : basic_string<charT>(str, n),
0, n, zero, one)
```

20.9.2.3 Members

```cpp
bitset<N>& operator&=(const bitset<N>& rhs) noexcept;
```

**Effects:** Clears each bit in *this for which the corresponding bit in \( \text{rhs} \) is clear, and leaves all other bits unchanged.

```cpp
bitset<N>& operator|=(const bitset<N>& rhs) noexcept;
```

**Effects:** Sets each bit in *this for which the corresponding bit in \( \text{rhs} \) is set, and leaves all other bits unchanged.
Returns: *this.

```cpp
bitset<N>& operator^=(const bitset<N>& rhs) noexcept;
```

Effects: Toggles each bit in *this for which the corresponding bit in rhs is set, and leaves all other bits unchanged.

Returns: *this.

```cpp
bitset<N>& operator<<=(size_t pos) noexcept;
```

Effects: Replaces each bit at position I in *this with a value determined as follows:

1. If I < pos, the new value is zero;
2. If I >= pos, the new value is the previous value of the bit at position I - pos.

Returns: *this.

```cpp
bitset<N>& operator>>=(size_t pos) noexcept;
```

Effects: Replaces each bit at position I in *this with a value determined as follows:

1. If pos >= N - I, the new value is zero;
2. If pos < N - I, the new value is the previous value of the bit at position I + pos.

Returns: *this.

```cpp
bitset<N>& set() noexcept;
```

Effects: Sets all bits in *this.

Returns: *this.

```cpp
bitset<N>& set(size_t pos, bool val = true);
```

Effects: Stores a new value in the bit at position pos in *this. If val is true, the stored value is one, otherwise it is zero.

Returns: *this.

Throws: out_of_range if pos does not correspond to a valid bit position.

```cpp
bitset<N>& reset() noexcept;
```

Effects: Resets all bits in *this.

Returns: *this.

```cpp
bitset<N>& reset(size_t pos);
```

Effects: Resets the bit at position pos in *this.

Returns: *this.

Throws: out_of_range if pos does not correspond to a valid bit position.

```cpp
bitset<N> operator~() const noexcept;
```

Effects: Constructs an object x of class bitset<N> and initializes it with *this.

Returns: x.flip().

```cpp
bitset<N>& flip() noexcept;
```

Effects: Toggles all bits in *this.

Returns: *this.

```cpp
bitset<N>& flip(size_t pos);
```

Effects: Toggles the bit at position pos in *this.

Returns: *this.

Throws: out_of_range if pos does not correspond to a valid bit position.
unsigned long to_ulong() const;

Returns: x.

Throws: overflow_error if the integral value x corresponding to the bits in *this cannot be represented as type unsigned long.

unsigned long long to_ullong() const;

Returns: x.

Throws: overflow_error if the integral value x corresponding to the bits in *this cannot be represented as type unsigned long long.

template<class charT = char,
    class traits = char_traits<charT>,
    class Allocator = allocator<charT>>
    basic_string<charT, traits, Allocator>
to_string(charT zero = charT('0'), charT one = charT('1')) const;

Effects: Constructs a string object of the appropriate type and initializes it to a string of length N characters. Each character is determined by the value of its corresponding bit position in *this. Character position N - 1 corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions. Bit value zero becomes the character zero, bit value one becomes the character one.

Returns: The created object.

size_t count() const noexcept;

Returns: A count of the number of bits set in *this.

constexpr size_t size() const noexcept;

Returns: N.

bool operator==(const bitset<N>& rhs) const noexcept;

Returns: true if the value of each bit in *this equals the value of the corresponding bit in rhs.

bool test(size_t pos) const;

Returns: true if the bit at position pos in *this has the value one.

Throws: out_of_range if pos does not correspond to a valid bit position.

bool all() const noexcept;

Returns: count() == size().

bool any() const noexcept;

Returns: count() != 0.

bool none() const noexcept;

Returns: count() == 0.

bitset<N> operator<<=(size_t pos) const noexcept;

Returns: bitset<N>(*this) <<= pos.

bitset<N> operator>>=(size_t pos) const noexcept;

Returns: bitset<N>(*this) >>= pos.

constexpr bool operator[](size_t pos) const;

Preconditions: pos is valid.

Returns: true if the bit at position pos in *this has the value one, otherwise false.

Throws: Nothing.
bitset<N>::reference operator[](size_t pos);

**Preconditions:** pos is valid.

**Returns:** An object of type bitset<N>::reference such that (*this)[pos] == this->test(pos), and such that (*this)[pos] = val is equivalent to this->set(pos, val).

**Throws:** Nothing.

**Remarks:** For the purpose of determining the presence of a data race (6.9.2), any access or update through the resulting reference potentially accesses or modifies, respectively, the entire underlying bitset.

### 20.9.3 bitset hash support

```cpp
template<size_t N> struct hash<bitset<N>>;
```

The specialization is enabled (20.14.19).

### 20.9.4 bitset operators

```cpp
bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
```

**Returns:** bitset<N>(lhs) & rhs.

```cpp
bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
```

**Returns:** bitset<N>(lhs) | rhs.

```cpp
bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
```

**Returns:** bitset<N>(lhs) ^= rhs.

```cpp
template<class charT, class traits, size_t N>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
```

**A formatted input function (29.7.4.3).**

**Effects:** Extracts up to N characters from is. Stores these characters in a temporary object str of type basic_string<charT, traits>, then evaluates the expression x = bitset<N>(str). Characters are extracted and stored until any of the following occurs:

1. N characters have been extracted and stored;
2. end-of-file occurs on the input sequence;
3. the next input character is neither is.widen('0') nor is.widen('1') (in which case the input character is not extracted).

If N > 0 and no characters are stored in str, calls is.setstate(ios_base::failbit) (which may throw ios_base::failure (29.5.5.4)).

**Returns:** is.

```cpp
template<class charT, class traits, size_t N>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
```

**Returns:** os << x.template to_string<charT, traits, allocator<charT>>((use_facet<ctype<charT>>(os.getloc())).widen('0'),
use_facet<ctype<charT>>(os.getloc())).widen('1'))

(see 29.7.5.3).

### 20.10 Memory

#### 20.10.1 In general

Subclause 20.10 describes the contents of the header <memory> (20.10.2) and some of the contents of the header <cstdlib> (17.2.2).
20.10.2 Header <memory> synopsis

The header <memory> defines several types and function templates that describe properties of pointers and pointer-like types, manage memory for containers and other template types, destroy objects, and construct objects in uninitialized memory buffers (20.10.3–20.10.11 and 25.11). The header also defines the templates unique_ptr, shared_ptr, weak_ptr, and various function templates that operate on objects of these types (20.11).

```cpp
#include <compare> // see 17.11.1

namespace std {
    // 20.10.3, pointer traits
    template<class Ptr> struct pointer_traits;
    template<class T> struct pointer_traits<T*>;

    // 20.10.4, pointer conversion
    template<class T> constexpr T* to_address(T* p) noexcept;
    template<class Ptr> constexpr auto to_address(const Ptr& p) noexcept;

    // 20.10.5, pointer safety
    enum class pointer_safety { relaxed, preferred, strict }
    void declare_reachable(void* p);
    template<class T> T* undeclare_reachable(T* p);
    void declare_no_pointers(char* p, size_t n);
    void undeclare_no_pointers(char* p, size_t n);
    pointer_safety get_pointer_safety() noexcept;

    // 20.10.6, pointer alignment
    void* align(size_t alignment, size_t size, void*& ptr, size_t& space);
    template<size_t N, class T> [[nodiscard]] constexpr T* assume_aligned(T* ptr);

    // 20.10.7, allocator argument tag
    struct allocator_arg_t { explicit allocator_arg_t() = default; }
    inline constexpr allocator_arg_t allocator_arg{};

    // 20.10.8, uses_allocator
    template<class T, class Alloc> struct uses_allocator;

    // 20.10.8.1, uses_allocator construction
    template<class T, class Alloc, class... Args>
    constexpr auto uses_allocator_construction_args(const Alloc& alloc,
        Args&&... args) noexcept -> see below;

    template<class T, class Alloc, class Tuple1, class Tuple2>
    constexpr auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t,
        Tuple1&& x, Tuple2&& y)
        noexcept -> see below;

    template<class T, class Alloc, class U, class V>
    constexpr auto uses_allocator_construction_args(const Alloc& alloc,
        const pair<U,V>&& pr) noexcept -> see below;

    template<class T, class Alloc, class U, class V>
    constexpr auto uses_allocator_construction_args(const Alloc& alloc,
        const pair<U,V>&& pr) noexcept -> see below;
```

§ 20.10.2
template<class T, class Alloc, class... Args>
constexpr T make_obj_using_allocator(const Alloc& alloc, Args&&... args);

template<class T, class Alloc, class... Args>
constexpr T* uninitialized_construct_using_allocator(T* p, const Alloc& alloc, Args&&... args);

// 20.10.9, allocator traits
template<class Alloc> struct allocator_traits;

// 20.10.10, the default allocator
template<class T> class allocator;

// 20.10.11, addressof
template<class T> constexpr T* addressof(T& r) noexcept;

template<class T> const T* addressof(const T&&) = delete;

// 25.11, specialized algorithms
// 25.11.2, special memory concepts

namespace ranges {
  template<class I, no-throw-input-range R> requires default_initializable<iter_value_t<I>>
    I uninitialized_default_construct(I first, S last);

  template<class ExecutionPolicy, class NoThrowForwardIterator> requires default_initializable<iter_value_t<ExecutionPolicy&>>
    void uninitialized_default_construct(ExecutionPolicy&& exec, NoThrowForwardIterator first, NoThrowForwardIterator last);

  template<class NoThrowForwardIterator, class Size> NoThrowForwardIterator
    uninitialized_default_construct_n(NoThrowForwardIterator first, Size n);

  template<class ExecutionPolicy, class NoThrowForwardIterator, class Size> NoThrowForwardIterator
    uninitialized_default_construct_n(ExecutionPolicy&& exec, NoThrowForwardIterator first, Size n);
}

template<class NoThrowForwardIterator> requires default_initializable<iter_value_t<NoThrowForwardIterator>>
  void uninitialized_value_construct(NoThrowForwardIterator first, NoThrowForwardIterator last);
template<class ExecutionPolicy, class NoThrowForwardIterator>
void uninitialized_value_construct(ExecutionPolicy&& exec, // see 25.3.5
NoThrowForwardIterator first,
NoThrowForwardIterator last);

template<class NoThrowForwardIterator, class Size>
NoThrowForwardIterator
uninitialized_value_construct_n(NoThrowForwardIterator first, Size n);

template<class ExecutionPolicy, class NoThrowForwardIterator, class Size>
NoThrowForwardIterator
uninitialized_value_construct_n(ExecutionPolicy&& exec, // see 25.3.5
NoThrowForwardIterator first, Size n);

namespace ranges {
    template<no-throw-forward-iterator I, no-throw-sentinel-for<I> S>
    requires default_initializable<iter_value_t<I>>
    I uninitialized_value_construct(I first, S last);
    template<no-throw-forward-range R>
    requires default_initializable<range_value_t<R>>
    borrowed_iterator_t<R> uninitialized_value_construct(R&& r);

    template<no-throw-forward-iterator I>
    requires default_initializable<iter_value_t<I>>
    I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
}

namespace ranges {
    template<class InputIterator, class NoThrowForwardIterator>
    NoThrowForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
    NoThrowForwardIterator result);
    template<class ExecutionPolicy, class InputIterator, class NoThrowForwardIterator>
    NoThrowForwardIterator uninitialized_copy(ExecutionPolicy&& exec, // see 25.3.5
    InputIterator first, InputIterator last,
    NoThrowForwardIterator result);
    template<class InputIterator, class Size, class NoThrowForwardIterator>
    NoThrowForwardIterator uninitialized_copy_n(InputIterator first, Size n,
    NoThrowForwardIterator result);
    template<class ExecutionPolicy, class InputIterator, class Size, class NoThrowForwardIterator>
    NoThrowForwardIterator uninitialized_copy_n(ExecutionPolicy&& exec, // see 25.3.5
    InputIterator first, Size n,
    NoThrowForwardIterator result);
}

namespace ranges {
    template<class I, class O>
    using uninitialized_copy_result = in_out_result<I, O>;
    template<input_iterator I, sentinel_for<I> S1,
    no-throw-forward-iterator O, no-throw-sentinel-for<O> S2>
    requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
    uninitialized_copy_result<I, O>
    uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);
    template<input_range IR, no-throw-forward-range OR>
    requires constructible_from<range_value_t<OR>, range_reference_t<IR>>
    uninitialized_copy_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
    uninitialized_copy(OR&& in_range, OR&& out_range);

    template<class I, class O>
    using uninitialized_copy_n_result = in_out_result<I, O>;
    template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel-for<O> S>
    requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
    uninitialized_copy_n_result<I, O>
    uninitialized_copy_n(I ifirst, iter_difference_t<I> n, 0 ofirst, S olast);
}

template<class InputIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_move(InputIterator first, InputIterator last,
NoThrowForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_move(ExecutionPolicy&& exec, // see 25.3.5
InputIterator first, InputIterator last,
NoThrowForwardIterator result);

template<class InputIterator, class Size, class NoThrowForwardIterator>
pair<InputIterator, NoThrowForwardIterator>
uninitialized_move_n(InputIterator first, Size n, NoThrowForwardIterator result);

template<class ExecutionPolicy, class InputIterator, class Size, class NoThrowForwardIterator>
pair<InputIterator, NoThrowForwardIterator>
uninitialized_move_n(ExecutionPolicy&& exec, // see 25.3.5
InputIterator first, Size n, NoThrowForwardIterator result);

namespace ranges {

template<class I, class O>
using uninitialized_move_result = in_out_result<I, O>;

template<input_iterator I, sentinel_for<I> S1,
no-throw-forward-iterator O, no-throw-sentinel-for<O> S2>
requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
uninitialized_move_result<I, O>
uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);

template<input_range IR, no-throw-forward-range OR>
requires constructible_from<range_value_t<OR>, range_rvalue_reference_t<IR>>
uninitialized_move_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
uninitialized_move(IR&& in_range, OR&& out_range);


template<class I, class O>
using uninitialized_move_n_result = in_out_result<I, O>;

template<input_iterator I,
no-throw-forward-iterator O,
no-throw-sentinel-for<O> S>
requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
uninitialized_move_n_result<I, O>
uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}


template<class NoThrowForwardIterator, class T>
void uninitialized_fill(NoThrowForwardIterator first, NoThrowForwardIterator last,
const T& x);

template<class ExecutionPolicy, class NoThrowForwardIterator, class T>
void uninitialized_fill(ExecutionPolicy&& exec, // see 25.3.5
NoThrowForwardIterator first, NoThrowForwardIterator last,
const T& x);

namespace ranges {

template<no-throw-forward_iterator I, no-throw-sentinel-for<I> S, class T>
requires constructible_from<iter_value_t<I>, const T&>
I uninitialized_fill(I first, S last, const T& x);

template<no-throw-forward-range R, class T>
requires constructible_from<range_value_t<R>, const T&>
borrowed_iterator_t<R> uninitialized_fill(R&& r, const T& x);


template<no-throw-forward_iterator I, class T>
requires constructible_from<iter_value_t<I>, const T&>
I uninitialized_fill_n(I first, iter_difference_t<I> n, const T& x);
}
// 25.11.8, construct_at
template<class T, class... Args>
constexpr T* construct_at(T* location, Args&&... args);

namespace ranges {
    template<class T, class... Args>
    constexpr T* construct_at(T* location, Args&&... args);
}

// 25.11.9, destroy
template<class T>
constexpr void destroy_at(T* location);

namespace ranges {
    template<class T, class... Args>
    constexpr T* construct_at(T* location, Args&&... args);
}

// 20.11.1, class template unique_ptr
template<class T> struct default_delete;

namespace ranges {
    template<class T, class D = default_delete<T>> class unique_ptr;
}

// 20.10.2 638

§ 20.10.2
template<class T1, class D1, class T2, class D2>
bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template<class T1, class D1, class T2, class D2>
bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template<class T1, class D1, class T2, class D2>
bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template<class T1, class D1, class T2, class D2>
bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template<class T1, class D1, class T2, class D2>
requires three_way_comparable_with<typename unique_ptr<T1, D1>::pointer,
    typename unique_ptr<T2, D2>::pointer>
    compare_three_way_result_t<typename unique_ptr<T1, D1>::pointer,
    typename unique_ptr<T2, D2>::pointer>
operator=>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template<class T, class D>
bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;

template<class T, class D>
bool operator<(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
bool operator<(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
bool operator>(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
bool operator>(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
bool operator<=(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
bool operator<=(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
requires three_way_comparable_with<typename unique_ptr<T, D>::pointer, nullptr_t>
    compare_three_way_result_t<typename unique_ptr<T, D>::pointer, nullptr_t>
operator<=>(const unique_ptr<T, D>& x, nullptr_t);

template<class E, class T, class Y, class D>
basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const unique_ptr<Y, D>& p);
// 20.11.2, class bad_weak_ptr
class bad_weak_ptr;

// 20.11.3, class template shared_ptr
template<class T> class shared_ptr;

// 20.11.3.7, shared_ptr creation
template<class T, class... Args>
shared_ptr<T> make_shared(Args&&... args); // T is not array

template<class T, class A, class... Args>
shared_ptr<T> allocate_shared(const A& a, Args&&... args); // T is not array

template<class T>
shared_ptr<T> make_shared(size_t N); // T is U[]

template<class T, class A>
shared_ptr<T> allocate_shared(const A& a, size_t N); // T is U[]

template<class T>
shared_ptr<T> make_shared(); // T is U[N]

template<class T, class A>
shared_ptr<T> allocate_shared(const A& a); // T is U[N]
template<class T>
    shared_ptr<T> make_shared(size_t N, const remove_extent_t<T>& u); // T is U[]

template<class T, class A>
    shared_ptr<T> allocate_shared(const A& a, size_t N,
        const remove_extent_t<T>& u); // T is U[]

template<class T>
    shared_ptr<T> make_shared(const remove_extent_t<T>& u); // T is U[]

template<class T, class A>
    shared_ptr<T> allocate_shared(const A& a, const remove_extent_t<T>& u); // T is U[]

// 20.11.3.8, shared_ptr comparisons
template<class T, class U>
    bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

template<class T, class U>
    strong_ordering operator<=>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

// 20.11.3.9, shared_ptr specialized algorithms
template<class T>
    void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.11.3.10, shared_ptr casts
template<class T, class U>
    shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;

template<class T, class U>
    static_pointer_cast(shared_ptr<U>&& r) noexcept;

template<class T, class U>
    dynamic_pointer_cast(const shared_ptr<U>&& r) noexcept;

template<class T, class U>
    dynamic_pointer_cast(shared_ptr<U>&& r) noexcept;

// 20.11.3.11, shared_ptr get_deleter
template<class D, class T>
    D* get_deleter(const shared_ptr<T>& p) noexcept;

// 20.11.3.12, shared_ptr I/O
enum class basic_ostream_type { E, T};

// 20.11.4, class template weak_ptr
template<class T> class weak_ptr;
// 20.11.4.7, weak_ptr specialized algorithms
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

// 20.11.5, class template owner_less
template<class T = void> struct owner_less;

// 20.11.6, class template enable_shared_from_this
template<class T> class enable_shared_from_this;

// 20.11.7, hash support
template<class T> struct hash;
template<class T, class D> struct hash<unique_ptr<T, D>>;
template<class T> struct hash<shared_ptr<T>>;

// 31.8.7, atomic smart pointers
template<class T> struct atomic;
template<class T> struct atomic<shared_ptr<T>>;
template<class T> struct atomic<weak_ptr<T>>;

20.10.3 Pointer traits

20.10.3.1 General

The class template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types.

```cpp
namespace std {
    template<class Ptr> struct pointer_traits {
        using pointer = Ptr;
        using element_type = see below;
        using difference_type = see below;
        template<class U> using rebind = see below;
        static pointer pointer_to(see below r);
    };
    template<class T> struct pointer_traits<T*> {
        using pointer = T*;
        using element_type = T;
        using difference_type = ptrdiff_t;
        template<class U> using rebind = U*;
        static constexpr pointer pointer_to(see below r) noexcept;
    };
}
```

20.10.3.2 Member types

```cpp
using element_type = see below;
```

1. Type: `Ptr::element_type` if the qualified-id `Ptr::element_type` is valid and denotes a type (13.10.3); otherwise, `T` if `Ptr` is a class template instantiation of the form `SomePointer<T, Args>`, where `Args` is zero or more type arguments; otherwise, the specialization is ill-formed.

```cpp
using difference_type = see below;
```

2. Type: `Ptr::difference_type` if the qualified-id `Ptr::difference_type` is valid and denotes a type (13.10.3); otherwise, `ptrdiff_t`.

```cpp
template<class U> using rebind = see below;
```

3. Alias template: `Ptr::rebind<U>` if the qualified-id `Ptr::rebind<U>` is valid and denotes a type (13.10.3); otherwise, `SomePointer<U, Args>` if `Ptr` is a class template instantiation of the form `SomePointer<T, Args>`, where `Args` is zero or more type arguments; otherwise, the instantiation of `rebind` is ill-formed.
20.10.3.3 Member functions

static pointer pointer_traits::pointer_to(see below r);
static constexpr pointer pointer_traits<T*>::pointer_to(see below r) noexcept;

1. Mandates: For the first member function, Ptr::pointer_to(r) is well-formed.
2. Preconditions: For the first member function, Ptr::pointer_to(r) returns a pointer to r through which indirection is valid.
3. Returns: The first member function returns Ptr::pointer_to(r). The second member function returns addressof(r).
4. Remarks: If element_type is cv void, the type of r is unspecified; otherwise, it is element_type&.

20.10.3.4 Optional members

Specializations of pointer_traits may define the member declared in this subclause to customize the behavior of the standard library.

static element_type* to_address(pointer p) noexcept;

1. Returns: A pointer of type element_type* that references the same location as the argument p.
2. [Note 1: This function is intended to be the inverse of pointer_to. If defined, it customizes the behavior of the non-member function to_address (20.10.4). — end note]

20.10.4 Pointer conversion

template<class T> constexpr T* to_address(T* p) noexcept;

1. Mandates: T is not a function type.

template<class Ptr> constexpr auto to_address(const Ptr& p) noexcept;

3. Returns: pointer_traits<Ptr>::to_address(p) if that expression is well-formed (see 20.10.3.4), otherwise to_address(p.operator->()).

20.10.5 Pointer safety

A complete object is declared reachable while the number of calls to declare_reachable with an argument referencing the object exceeds the number of calls to undeclare_reachable with an argument referencing the object.

void declare_reachable(void* p);

1. Preconditions: p is a safely-derived pointer (6.7.5.5.4) or a null pointer value.
2. Effects: If p is not null, the complete object referenced by p is subsequently declared reachable (6.7.5.5.4).
3. Throws: May throw bad_alloc if the system cannot allocate additional memory that may be required to track objects declared reachable.

template<class T> T* undeclare_reachable(T* p);

5. Preconditions: If p is not null, the complete object referenced by p has been previously declared reachable, and is live (6.7.3) from the time of the call until the last undeclare_reachable(p) call on the object.

[Note 1: It is expected that calls to declare_reachable(p) consume a small amount of memory in addition to that occupied by the referenced object until the matching call to undeclare_reachable(p) is encountered. Thus, long-running programs where calls are not matched can exhibit a memory leak. — end note]

void declare_no_pointers(char* p, size_t n);

9. Preconditions: No bytes in the specified range are currently registered with declare_no_pointers(). If the specified range is in an allocated object, then it is entirely within a single allocated object. The object is live until the corresponding undeclare_no_pointers() call.
Note 2: In a garbage-collecting implementation, the fact that a region in an object is registered with `declare_no_pointers()` does not prevent the object from being collected. — end note

Effects: The n bytes starting at p no longer contain traceable pointer locations, independent of their type. Hence indirection through a pointer located there is undefined if the object it points to was created by global `operator new` and not previously declared reachable.

Note 3: This can be used to inform a garbage collector or leak detector that this region of memory need not be traced. — end note

Throws: Nothing.

Note 4: The request can be ignored if a memory allocation needed by the implementation fails. — end note

void undeclare_no_pointers(char* p, size_t n);

Preconditions: The same range has previously been passed to `declare_no_pointers()`.

Effects: Unregisters a range registered with `declare_no_pointers()` for destruction. It shall be called before the lifetime of the object ends.

Throws: Nothing.

`pointer_safety get_pointer_safety()` noexcept;

Returns: `pointer_safety::strict` if the implementation has strict pointer safety (6.7.5.5.4). It is implementation-defined whether `get_pointer_safety` returns `pointer_safety::relaxed` or `pointer_safety::preferred` if the implementation has relaxed pointer safety.

20.10.6 Pointer alignment

void* align(size_t alignment, size_t size, void*& ptr, size_t& space);

Preconditions:

(1.1) alignment is a power of two

(1.2) ptr represents the address of contiguous storage of at least space bytes

Effects: If it is possible to fit size bytes of storage aligned by alignment into the buffer pointed to by ptr with length space, the function updates ptr to represent the first possible address of such storage and decreases space by the number of bytes used for alignment. Otherwise, the function does nothing.

Returns: A null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of ptr.

Note 1: The function updates its ptr and space arguments so that it can be called repeatedly with possibly different alignment and size arguments for the same buffer. — end note

template<size_t N, class T>
[[nodiscard]] constexpr T* assume_aligned(T* ptr);

Mandates: N is a power of two.

Preconditions: ptr points to an object X of a type similar (7.3.6) to T, where X has alignment N (6.7.6).

Returns: ptr.

Throws: Nothing.

Note 2: The alignment assumption on an object X expressed by a call to `assume_aligned` might result in generation of more efficient code. It is up to the program to ensure that the assumption actually holds. The call does not cause the compiler to verify or enforce this. An implementation might only make the assumption for those operations on X that access X through the pointer returned by `assume_aligned`. — end note

20.10.7 Allocator argument tag

namespace std {
  struct allocator_arg_t { explicit allocator_arg_t() = default; };
  inline constexpr allocator_arg_t allocator_arg{};
}

224) `pointer_safety::preferred` might be returned to indicate that a leak detector is running so that the program can avoid spurious leak reports.
The allocator_arg_t struct is an empty class type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see tuple 20.5) have constructors with allocator_arg_t as the first argument, immediately followed by an argument of a type that meets the Cpp17Allocator requirements (Table 36).

20.10.8 uses_allocator

20.10.8.1 uses_allocator trait

```cpp
template<class T, class Alloc> struct uses_allocator;
```

Remarks: Automatically detects whether T has a nested allocator_type that is convertible from Alloc. Meets the Cpp17BinaryTypeTrait requirements (20.15.2). The implementation shall provide a definition that is derived from true_type if the qualified-id T::allocator_type is valid and denotes a type (13.10.3) and is_convertible_v<Alloc, T::allocator_type> != false, otherwise it shall be derived from false_type. A program may specialize this template to derive from true_type for a program-defined type T that does not have a nested allocator_type but nonetheless can be constructed with an allocator where either:

(1.1) the first argument of a constructor has type allocator_arg_t and the second argument has type Alloc or

(1.2) the last argument of a constructor has type Alloc.

20.10.8.2 Uses-allocator construction

Uses-allocator construction with alloc and constructor arguments args... refers to the construction of an object of type T such that alloc is passed to the constructor of T if T uses an allocator type compatible with alloc. When applied to the construction of an object of type T, it is equivalent to initializing it with the value of the expression make_obj_using_allocator<T>(alloc, args...), described below.

The following utility functions support three conventions for passing alloc to a constructor:

(2.1) If T does not use an allocator compatible with alloc, then alloc is ignored.

(2.2) Otherwise, if T has a constructor invocable as T(allocator_arg, alloc, args...) (leading-allocator convention), then uses-allocator construction chooses this constructor form.

(2.3) Otherwise, if T has a constructor invocable as T(args..., alloc) (trailing-allocator convention), then uses-allocator construction chooses this constructor form.

The uses_allocator_construction_args function template takes an allocator and argument list and produces (as a tuple) a new argument list matching one of the above conventions. Additionally, overloads are provided that treat specializations of pair such that uses-allocator construction is applied individually to the first and second data members. The make_obj_using_allocator and uninitialized_construct_using_allocator function templates apply the modified constructor arguments to construct an object of type T as a return value or in-place, respectively.

[Note 1: For uses_allocator_construction_args and make_obj_using_allocator, type T is not deduced and must therefore be specified explicitly by the caller. — end note]
[Note 2: This definition prevents a silent failure to pass the allocator to a constructor of a type for which uses_allocator_v<T, Alloc> is true. — end note]

template<class T, class Alloc, class Tuple1, class Tuple2>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t, Tuple1&& x, Tuple2&& y)
    noexcept -> see below;

Constraints: T is a specialization of pair.
Effects: For T specified as pair<T1, T2>, equivalent to:
    return make_tuple(  
        piecewise_construct,  
        apply([&alloc](auto&&... args1) {  
            return uses_allocator_construction_args<T1>(  
                alloc, std::forward<decltype(args1)>(args1)...);  
        }, std::forward<Tuple1>(x)),  
        apply([&alloc](auto&&... args2) {  
            return uses_allocator_construction_args<T2>(  
                alloc, std::forward<decltype(args2)>(args2)...);  
        }, std::forward<Tuple2>(y)));

template<class T, class Alloc>
constexpr auto uses_allocator_construction_args(const Alloc& alloc) noexcept -> see below;

Constraints: T is a specialization of pair.
Effects: Equivalent to:
    return uses_allocator_construction_args<T>({allocate, piecewise_construct, tuple<>()}, tuple<>());

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, U&& u, V&& v) noexcept -> see below;

Constraints: T is a specialization of pair.
Effects: Equivalent to:
    return uses_allocator_construction_args<T>({allocate, piecewise_construct,  
        forward_as_tuple(std::forward<U>(u)),  
        forward_as_tuple(std::forward<V>(v))});

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, const pair<U,V>& pr) noexcept -> see below;

Constraints: T is a specialization of pair.
Effects: Equivalent to:
    return uses_allocator_construction_args<T>({allocate, piecewise_construct,  
        forward_as_tuple(pr.first),  
        forward_as_tuple(pr.second)});

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, pair<U,V>&& pr) noexcept -> see below;

Constraints: T is a specialization of pair.
Effects: Equivalent to:
    return uses_allocator_construction_args<T>({allocate, piecewise_construct,  
        forward_as_tuple(std::move(pr).first),  
        forward_as_tuple(std::move(pr).second)});

template<class T, class Alloc, class... Args>
constexpr T make_obj_using_allocator(const Alloc& alloc, Args&&... args);

Effects: Equivalent to:
return make_from_tuple<T>(uses_allocator_construction_args<T>(
    alloc, std::forward<Args>(args)...));

template<class T, class Alloc, class... Args>
constexpr T* uninitialized_construct_using_allocator(T* p, const Alloc& alloc, Args&&... args);

Effects: Equivalent to:
return apply([&]<class... U>(U&&... xs) {
    return construct_at(p, std::forward<U>(xs)...);
}, uses_allocator_construction_args<T>(alloc, std::forward<Args>(args)...));

20.10.9 Allocator traits

20.10.9.1 General

The class template allocator_traits supplies a uniform interface to all allocator types. An allocator cannot
be a non-class type, however, even if allocator_traits supplies the entire required interface.

[Note 1: Thus, it is always possible to create a derived class from an allocator. — end note]

namespace std {
    template<class Alloc> struct allocator_traits {
        using allocator_type = Alloc;
        using value_type = typename Alloc::value_type;
        using pointer = see below;
        using const_pointer = see below;
        using void_pointer = see below;
        using const_void_pointer = see below;
        using difference_type = see below;
        using size_type = see below;
        using propagate_on_container_copy_assignment = see below;
        using propagate_on_container_move_assignment = see below;
        using propagate_on_container_swap = see below;
        using is_always_equal = see below;
        template<class T> using rebind_alloc = see below;
        template<class T> using rebind_traits = allocator_traits<rebind_alloc<T>>;

        [[nodiscard]] static constexpr pointer allocate(Alloc& a, size_type n);
        [[nodiscard]] static constexpr pointer allocate(Alloc& a, size_type n,
            const_void_pointer hint);
        static constexpr void deallocate(Alloc& a, pointer p, size_type n);
        template<class T, class... Args>
            static constexpr void construct(Alloc& a, T* p, Args&&... args);
        template<class T>
            static constexpr void destroy(Alloc& a, T* p);
        static constexpr size_type max_size(const Alloc& a) noexcept;
        static constexpr Alloc select_on_container_copy_construction(const Alloc& rhs);
    };
}

20.10.9.2 Member types

using pointer = see below;

Type: Alloc::pointer if the qualified-id Alloc::pointer is valid and denotes a type (13.10.3);
otherwise, value_type*.
using const_pointer = see below;

2    Type: Alloc::const_pointer if the qualified-id Alloc::const_pointer is valid and denotes a type (13.10.3); otherwise, pointer_traits<pointer>::rebind<const value_type>.

using void_pointer = see below;

3    Type: Alloc::void_pointer if the qualified-id Alloc::void_pointer is valid and denotes a type (13.10.3); otherwise, pointer_traits<pointer>::rebind<void>.

using const_void_pointer = see below;

4    Type: Alloc::const_void_pointer if the qualified-id Alloc::const_void_pointer is valid and denotes a type (13.10.3); otherwise, pointer_traits<pointer>::rebind<const void>.

using difference_type = see below;

5    Type: Alloc::difference_type if the qualified-id Alloc::difference_type is valid and denotes a type (13.10.3); otherwise, pointer_traits<pointer>::difference_type.

using size_type = see below;

6    Type: Alloc::size_type if the qualified-id Alloc::size_type is valid and denotes a type (13.10.3); otherwise, make_unsigned_t<difference_type>.

using propagate_on_container_copy_assignment = see below;

7    Type: Alloc::propagate_on_container_copy_assignment if the qualified-id Alloc::propagate_on_container_copy_assignment is valid and denotes a type (13.10.3); otherwise false_type.

using propagate_on_container_move_assignment = see below;

8    Type: Alloc::propagate_on_container_move_assignment if the qualified-id Alloc::propagate_on_container_move_assignment is valid and denotes a type (13.10.3); otherwise false_type.

using propagate_on_container_swap = see below;

9    Type: Alloc::propagate_on_container_swap if the qualified-id Alloc::propagate_on_container_swap is valid and denotes a type (13.10.3); otherwise false_type.

using is_always_equal = see below;

10   Type: Alloc::is_always_equal if the qualified-id Alloc::is_always_equal is valid and denotes a type (13.10.3); otherwise is_empty<Alloc>::type.

template<class T> using rebind_alloc = see below;

11   Alias template: Alloc::rebind<T>::other if the qualified-id Alloc::rebind<T>::other is valid and denotes a type (13.10.3); otherwise Alloc<T, Args> if Alloc is a class template instantiation of the form Alloc<U, Args>, where Args is zero or more type arguments; otherwise, the instantiation of rebind_alloc is ill-formed.

20.10.9.3 Static member functions

[[nodiscard]] static constexpr pointer allocate(Alloc& a, size_type n);

Returns: a.allocate(n).

1

[[nodiscard]] static constexpr pointer allocate(Alloc& a, size_type n, const_void_pointer hint);

2    Returns: a.allocate(n, hint) if that expression is well-formed; otherwise, a.allocate(n).

static constexpr void deallocate(Alloc& a, pointer p, size_type n);

3    Effects: Calls a.deallocate(p, n).

4    Throws: Nothing.

template<class T, class... Args>

5    static constexpr void deallocate(Alloc& a, T* p, Args&&... args);

6    Effects: Calls a.deallocate(p, std::forward<Args>(args)... ) if that call is well-formed; otherwise, invokes construct_at(p, std::forward<Args>(args)... ).
template<class T>
static constexpr void destroy(Alloc& a, T* p);

Effects: Calls a.destroy(p) if that call is well-formed; otherwise, invokes destroy_at(p).

static constexpr size_type max_size(const Alloc& a) noexcept;

Returns: a.max_size() if that expression is well-formed; otherwise, numeric_limits<size_type>::max()/sizeof(value_type).

static constexpr Alloc select_on_container_copy_construction(const Alloc& rhs);

Returns: rhs.select_on_container_copy_construction() if that expression is well-formed; otherwise, rhs.

20.10.10 The default allocator

20.10.10.1 General

All specializations of the default allocator meet the allocator completeness requirements (16.4.4.6.2).

namespace std {
    template<class T> class allocator {
        public:
            using value_type = T;
            using size_type = size_t;
            using difference_type = ptrdiff_t;
            using propagate_on_container_move_assignment = true_type;
            using is_always_equal = true_type;
            constexpr allocator() noexcept;
            constexpr allocator(const allocator&) noexcept;
            template<class U> constexpr allocator(const allocator<U>&) noexcept;
            constexpr ~allocator();
            constexpr allocator& operator=(const allocator&) = default;
            [[nodiscard]] constexpr T* allocate(size_t n);
            constexpr void deallocate(T* p, size_t n);
    }
}

20.10.10.2 Members

Except for the destructor, member functions of the default allocator shall not introduce data races (6.9.2) as a result of concurrent calls to those member functions from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

[[nodiscard]] constexpr T* allocate(size_t n);

Mandates: T is not an incomplete type (6.8).

Returns: A pointer to the initial element of an array of n T.

Throws: bad_array_new_length if numeric_limits<size_t>::max() / sizeof(T) < n, or bad_alloc if the storage cannot be obtained.

Remarks: The storage for the array is obtained by calling ::operator new (17.6.3), but it is unspecified when or how often this function is called. This function starts the lifetime of the array object, but not that of any of the array elements.

constexpr void deallocate(T* p, size_t n);

Preconditions: p is a pointer value obtained from allocate(). n equals the value passed as the first argument to the invocation of allocate which returned p.

Effects: Deallocates the storage referenced by p.

Remarks: Uses ::operator delete (17.6.3), but it is unspecified when this function is called.
20.10.10.3 Operators

```cpp
template<class T, class U>
constexpr bool operator==(const allocator<T>&, const allocator<U>&) noexcept;
```

1

**Returns:** true.

20.10.11 `addressof`

```cpp
template<class T> constexpr T* addressof(T& r) noexcept;
```

1

**Returns:** The actual address of the object or function referenced by r, even in the presence of an overloaded `operator&`.

2

**Remarks:** An expression `addressof(E)` is a constant subexpression (3.14) if E is an lvalue constant subexpression.

20.10.12 C library memory allocation

```
void* aligned_alloc(size_t alignment, size_t size);
void* calloc(size_t nmemb, size_t size);
void* malloc(size_t size);
void* realloc(void* ptr, size_t size);
```

2

**Effects:** These functions have the semantics specified in the C standard library.

3

**Remarks:** These functions do not attempt to allocate storage by calling `operator new()` (17.6.3).

4

Storage allocated directly with these functions is implicitly declared reachable (see 6.7.5.5.4) on allocation, ceases to be declared reachable on deallocation, and need not cease to be declared reachable as the result of an `undeclare_reachable()` call.

[Note 1: The header `<cstdlib>` (17.2.2) declares the functions described in this subclause. —end note]

```
void free(void* ptr);
```

6

**Effects:** This function has the semantics specified in the C standard library.

7

**Remarks:** This function does not attempt to deallocate storage by calling `operator delete()`.

See also: ISO C 7.22.3

20.11 Smart pointers

20.11.1 Class template `unique_ptr`

```cpp
20.11.1.1 General
```

1

A `unique pointer` is an object that owns another object and manages that other object through a pointer. More precisely, a unique pointer is an object u that stores a pointer to a second object p and will dispose of p when u is itself destroyed (e.g., when leaving block scope (8.8)). In this context, u is said to own p.

2

The mechanism by which u disposes of p is known as p’s associated deleter, a function object whose correct invocation results in p’s appropriate disposition (typically its deletion).

3

Let the notation u.p denote the pointer stored by u, and let u.d denote the associated deleter. Upon request, u can reset (replace) u.p and u.d with another pointer and deleter, but properly disposes of its owned object via the associated deleter before such replacement is considered completed.

4

Each object of a type U instantiated from the `unique_ptr` template specified in 20.11.1 has the strict ownership semantics, specified above, of a unique pointer. In partial satisfaction of these semantics, each such U is `Cpp17MoveConstructible` and `Cpp17MoveAssignale`, but is not `Cpp17CopyConstructible` nor `Cpp17CopyAssignable`. The template parameter T of `unique_ptr` may be an incomplete type.
20.11.1.2 Default deleters

20.11.1.2.1 In general

The class template `default_delete` serves as the default deleter (destruction policy) for the class template `unique_ptr`.

The template parameter `T` of `default_delete` may be an incomplete type.

### 20.11.1.2.2 default_delete

```cpp
namespace std {
    template<class T> struct default_delete {
        constexpr default_delete() noexcept = default;
        template<class U> default_delete(const default_delete<U>&) noexcept;
        void operator()(T*) const;
    };
}
```

- **Constraints**: `U*` is implicitly convertible to `T*`.
- **Effects**: Constructs a `default_delete` object from another `default_delete<U>` object.
- **Mandates**: `T` is a complete type.
- **Effects**: Calls `delete` on `ptr`.

### 20.11.1.2.3 default_delete<T[]>

```cpp
namespace std {
    template<class T> struct default_delete<T[]> {
        constexpr default_delete() noexcept = default;
        template<class U> default_delete(const default_delete<U[]>& other) noexcept;
        template<class U> void operator()(U* ptr) const;
    };
}
```

- **Constraints**: `U(*)[]` is convertible to `T(*)[]`.
- **Effects**: Constructs a `default_delete` object from another `default_delete<U[]>` object.
- **Mandates**: `U(*)[]` is convertible to `T(*)[]`.
- **Effects**: Calls `delete[]` on `ptr`.

20.11.1.3 `unique_ptr` for single objects

20.11.1.3.1 General

```cpp
namespace std {
    template<class T, class D = default_delete<T>> class unique_ptr {
        public:
            using pointer = see below;
            using element_type = T;
            using deleter_type = D;

            // 20.11.1.3.2, constructors
            constexpr unique_ptr() noexcept;
            explicit unique_ptr(pointer p) noexcept;
            unique_ptr(pointer p, see below d1) noexcept;
    }
}
```
unique_ptr(pointer p, see below d2) noexcept;
unique_ptr(unique_ptr&& u) noexcept;
constexpr unique_ptr(nullptr_t) noexcept;
template<class U, class E>
  unique_ptr(unique_ptr<U, E>&& u) noexcept;

// 20.11.1.3.3, destructor
~unique_ptr();

// 20.11.1.3.4, assignment
unique_ptr& operator=(unique_ptr&& u) noexcept;
template<class U, class E>
  unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
unique_ptr& operator=(nullptr_t) noexcept;

// 20.11.1.3.5, observers
add_lvalue_reference_t<T> operator*() const;
pointer operator->() const noexcept;
pointer get() const noexcept;
deleter_type& get_deleter() noexcept;
const deleter_type& get_deleter() const noexcept;
explicit operator bool() const noexcept;

// 20.11.1.3.6, modifiers
pointer release() noexcept;
void reset(pointer p = pointer()) noexcept;
void swap(unique_ptr& u) noexcept;

// disable copy from lvalue
unique_ptr(const unique_ptr&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;
}

1 The default type for the template parameter D is default_delete. A client-supplied template argument D
shall be a function object type (20.14), lvalue reference to function, or lvalue reference to function object type
for which, given a value d of type D and a value ptr of type unique_ptr<T, D>::pointer, the expression
  d(ptr)
is valid and has the effect of disposing of the pointer as appropriate for that deleter.

2 If the deleter’s type D is not a reference type, D shall meet the Cpp17Destructible requirements (Table 32).

3 If the qualified-id remove_reference_t<D>::pointer is valid and denotes a type (13.10.3), then unique_ptr<T, D>::pointer
shall be a synonym for remove_reference_t<D>::pointer. Otherwise unique_ptr<T, D>::pointer shall be a synonym for element_type*. The type unique_ptr<T, D>::pointer shall meet
the Cpp17NullablePointer requirements (Table 33).

4 [Example 1: Given an allocator type X (Table 36) and letting A be a synonym for allocator_traits<X>, the types
A::pointer, A::const_pointer, A::void_pointer, and A::const_void_pointer may be used as unique_ptr<T, D>::pointer. — end example]

20.11.1.3.2 Constructors

constexpr unique_ptr() noexcept;
constexpr unique_ptr(nullptr_t) noexcept;

1 Constraints: is_pointer_v<deleter_type> is false and is_default_constructible_v<deleter_type> is true.

2 Preconditions: D meets the Cpp17DefaultConstructible requirements (Table 27), and that construction
does not throw an exception.

3 Effects: Constructs a unique_ptr object that owns nothing, value-initializing the stored pointer and
the stored deleter.

4 Postconditions: get() == nullptr. get_deleter() returns a reference to the stored deleter.
explicit unique_ptr(pointer p) noexcept;

Constraints: is_pointer_v<deleter_type> is false and is_default_constructible_v<deleter_type> is true.

Mandates: This constructor is not selected by class template argument deduction (12.4.2.9).

Preconditions: D meets the Cpp17DefaultConstructible requirements (Table 27), and that construction does not throw an exception.

Effects: Constructs a unique_ptr which owns p, initializing the stored pointer with p and value-initializing the stored deleter.

Postconditions: get() == p. get_deleter() returns a reference to the stored deleter.

unique_ptr(pointer p, const D& d) noexcept;
unique_ptr(pointer p, remove_reference_t<D>&& d) noexcept;

Constraints: is_constructible_v<D, decltype(d)> is true.

Mandates: These constructors are not selected by class template argument deduction (12.4.2.9).

Preconditions: For the first constructor, if D is not a reference type, D meets the Cpp17CopyConstructible requirements and such construction does not exit via an exception. For the second constructor, if D is not a reference type, D meets the Cpp17MoveConstructible requirements and such construction does not exit via an exception.

Effects: Constructs a unique_ptr object which owns p, initializing the stored pointer with p and initializing the deleter from std::forward<decltype(d)>(d).

Postconditions: get() == p. get_deleter() returns a reference to the stored deleter. If D is a reference type then get_deleter() returns a reference to the lvalue d.

Remarks: If D is a reference type, the second constructor is defined as deleted.

[Example 1:

D d;
unique_ptr<int, D> p1(new int, D()); // D must be Cpp17MoveConstructible
unique_ptr<int, D> p2(new int, d); // D must be Cpp17CopyConstructible
unique_ptr<int, D&> p3(new int, d); // p3 holds a reference to d
unique_ptr<int, const D&> p4(new int, D()); // error: rvalue deleter object combined
// with reference deleter type

— end example]

unique_ptr(unique_ptr&& u) noexcept;

Constraints: is_move_constructible_v<D> is true.

Preconditions: If D is not a reference type, D meets the Cpp17MoveConstructible requirements (Table 28). Construction of the deleter from an rvalue of type D does not throw an exception.

Effects: Constructs a unique_ptr from u. If D is a reference type, this deleter is copy constructed from u’s deleter; otherwise, this deleter is move constructed from u’s deleter.

[Note 1: The construction of the deleter can be implemented with std::forward<>. — end note]

Postconditions: get() yields the value u.get() yielded before the construction. u.get() == nullptr. get_deleter() returns a reference to the stored deleter that was constructed from u.get_deleter(). If D is a reference type then get_deleter() and u.get_deleter() both reference the same lvalue deleter.

template<class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;

Constraints:

(21.1) unique_ptr<U, E>::pointer is implicitly convertible to pointer,
(21.2) U is not an array type, and
(21.3) either D is a reference type and E is the same type as D, or D is not a reference type and E is implicitly convertible to D.
Preconditions: If \( E \) is not a reference type, construction of the deleter from an rvalue of type \( E \) is well-formed and does not throw an exception. Otherwise, \( E \) is a reference type and construction of the deleter from an lvalue of type \( E \) is well-formed and does not throw an exception.

Effects: Constructs a `unique_ptr` from \( u \). If \( E \) is a reference type, this deleter is copy constructed from \( u \)'s deleter; otherwise, this deleter is move constructed from \( u \)'s deleter.

[Note 2: The deleter constructor can be implemented with `std::forward<E>`. — end note]

Postconditions: `get()` yields the value `u.get()` yielded before the construction. `u.get() == nullptr`. `get_deleter()` returns a reference to the stored deleter that was constructed from `u.get_deleter()`.

20.11.1.3.3 Destructor [unique.ptr.single.dtor]

```cpp
~unique_ptr();
```

Preconditions: The expression `get_deleter()(get())` is well-formed, has well-defined behavior, and does not throw exceptions.

[Note 1: The use of `default_delete` requires \( T \) to be a complete type. — end note]

Effects: If `get() == nullptr` there are no effects. Otherwise `get_deleter()(get())`.

20.11.1.3.4 Assignment [unique.ptr.single.asgn]

```cpp
unique_ptr& operator=(unique_ptr&& u) noexcept;
```

Constraints: `is_move_assignable_v<D>` is true.

Preconditions: If \( D \) is not a reference type, \( D \) meets the `Cpp17MoveAssignable` requirements (Table 30) and assignment of the deleter from an rvalue of type \( D \) does not throw an exception. Otherwise, \( D \) is a reference type; `remove_reference_t<D>` meets the `Cpp17CopyAssignable` requirements and assignment of the deleter from an lvalue of type \( D \) does not throw an exception.

Effects: Calls `reset(u.release())` followed by `get_deleter() = std::forward<D>(u.get_deleter())`.

Postconditions: `u.get() == nullptr`.

Returns: `*this`.

```cpp
template<class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
```

Constraints:

(6.1) unique_ptr\(<U, E>::pointer\) is implicitly convertible to `pointer`, and

(6.2) \( U \) is not an array type, and

(6.3) `is_assignable_v<D&, E&&>` is true.

Preconditions: If \( E \) is not a reference type, assignment of the deleter from an rvalue of type \( E \) is well-formed and does not throw an exception. Otherwise, \( E \) is a reference type and assignment of the deleter from an lvalue of type \( E \) is well-formed and does not throw an exception.

Effects: Calls `reset(u.release())` followed by `get_deleter() = std::forward<E>(u.get_deleter())`.

Postconditions: `u.get() == nullptr`.

Returns: `*this`.

20.11.1.3.5 Observers [unique.ptr.single.observers]

```cpp
add_lvalue_reference_t<T> operator*() const;
```

Preconditions: `get() != nullptr`.

Returns: `*get()`.
pointer operator->() const noexcept;

3  Preconditions: get() != nullptr.
4  Returns: get().

5  [Note 1: The use of this function typically requires that T be a complete type. —end note]

pointer get() const noexcept;

6  Returns: The stored pointer.

deleter_type& get_deleter() noexcept;
const deleter_type& get_deleter() const noexcept;

7  Returns: A reference to the stored deleter.

explicit operator bool() const noexcept;

8  Returns: get() != nullptr.

20.11.1.3.6 Modifiers

pointer release() noexcept;

1  Postconditions: get() == nullptr.
2  Returns: The value get() had at the start of the call to release.

void reset(pointer p = pointer()) noexcept;

3  Preconditions: The expression get_deleter()(get()) is well-formed, has well-defined behavior, and
does not throw exceptions.
4  Effects: Assigns p to the stored pointer, and then if and only if the old value of the stored pointer,
old_p, was not equal to nullptr, calls get_deleter()(old_p).

[Note 1: The order of these operations is significant because the call to get_deleter() might destroy *this.
—end note]

5  Postconditions: get() == p.

[Note 2: The postcondition does not hold if the call to get_deleter() destroys *this since this->get() is no
longer a valid expression. —end note]

void swap(unique_ptr& u) noexcept;

6  Preconditions: get_deleter() is swappable (16.4.4.3) and does not throw an exception under swap.
7  Effects: Invokes swap on the stored pointers and on the stored deleters of *this and u.

20.11.1.4 unique_ptr for array objects with a runtime length

20.11.1.4.1 General

namespace std {
  template<class T, class D> class unique_ptr<T[], D> {
    public:
      using pointer = see below;
      using element_type = T;
      using deleter_type = D;

    // 20.11.1.4.2, constructors
    constexpr unique_ptr() noexcept;
    template<class U> explicit unique_ptr(U p) noexcept;
    template<class U> unique_ptr(U p, see below d) noexcept;
    template<class U> unique_ptr(U p, see below d) noexcept;
    unique_ptr(unique_ptr&& u) noexcept;
    template<class U, class E>
       unique_ptr(unique_ptr<U, E>&& u) noexcept;
    constexpr unique_ptr=nullptr_t) noexcept;

    // destructor
    ~unique_ptr();

§ 20.11.1.4.1
// assignment
unique_ptr& operator=(unique_ptr&& u) noexcept;

// 20.11.1.4.4, observers
t& operator[](size_t i) const;
pointer get() const noexcept;
deleter_type& get_deleter() noexcept;
const deleter_type& get_deleter() const noexcept;
explicit operator bool() const noexcept;

// 20.11.1.4.5, modifiers
pointer release() noexcept;
template<class U> void reset(U p) noexcept;
void reset(nullptr_t = nullptr) noexcept;
void swap(unique_ptr& u) noexcept;

// disable copy from lvalue
unique_ptr(const unique_ptr&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;

A specialization for array types is provided with a slightly altered interface.

1 Conversions between different types of unique_ptr<T[], D> that would be disallowed for the corresponding pointer-to-array types, and conversions to or from the non-array forms of unique_ptr, produce an ill-formed program.

2 Pointers to types derived from T are rejected by the constructors, and by reset.

3 The observers operator* and operator-> are not provided.

4 The indexing observer operator[] is provided.

5 The default deleter will call delete[].

2 Descriptions are provided below only for members that differ from the primary template.

The template argument T shall be a complete type.

20.11.1.4.2 Constructors

template<class U> explicit unique_ptr(U p) noexcept;

This constructor behaves the same as the constructor in the primary template that takes a single parameter of type pointer.

Constraints:

1 U is the same type as pointer, or

2 pointer is the same type as element_type*, U is a pointer type V*, and V(*)[] is convertible to element_type(*)([]).

template<class U> unique_ptr(U p, see below d) noexcept;
template<class U> unique_ptr(U p, see below d) noexcept;

3 These constructors behave the same as the constructors in the primary template that take a parameter of type pointer and a second parameter.

Constraints:

4 U is the same type as pointer,

5 U is nullptr_t, or

6 pointer is the same type as element_type*, U is a pointer type V*, and V(*)([]) is convertible to element_type(*)([]).
template<class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;

This constructor behaves the same as in the primary template.

Constraints: Where UP is unique_ptr<U, E>:

(6.1) — U is an array type, and
(6.2) — pointer is the same type as element_type*, and
(6.3) — UP::pointer is the same type as UP::element_type*, and
(6.4) — UP::element_type(*)[] is convertible to element_type(*)[], and
(6.5) — either D is a reference type and E is the same type as D, or D is not a reference type and E is
implicitly convertible to D.

[Note 1: This replaces the Constraints: specification of the primary template. — end note]

20.11.1.4.3 Assignment

template<class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;

This operator behaves the same as in the primary template.

Constraints: Where UP is unique_ptr<U, E>:

(2.1) — U is an array type, and
(2.2) — pointer is the same type as element_type*, and
(2.3) — UP::pointer is the same type as UP::element_type*, and
(2.4) — UP::element_type(*)[] is convertible to element_type(*)[], and
(2.5) — is_assignable_v<D&, E&&> is true.

[Note 1: This replaces the Constraints: specification of the primary template. — end note]

20.11.1.4.4 Observers

T& operator[](size_t i) const;

Preconditions: i < the number of elements in the array to which the stored pointer points.

Returns: get()[i].

20.11.1.4.5 Modifiers

void reset(nullptr_t p = nullptr) noexcept;

Effects: Equivalent to reset(pointer()).

template<class U> void reset(U p) noexcept;

This function behaves the same as the reset member of the primary template.

Constraints:

(3.1) — U is the same type as pointer, or
(3.2) — pointer is the same type as element_type*, U is a pointer type V*, and V(*)[] is convertible to
element_type(*)[].

20.11.1.5 Creation

template<class T, class... Args> unique_ptr<T> make_unique(Args&&... args);

Constraints: T is not an array type.

Returns: unique_ptr<T>(new T(std::forward<Args>(args)...)).

template<class T> unique_ptr<T> make_unique(size_t n);

Constraints: T is an array of unknown bound.

Returns: unique_ptr<T>(new remove_extent_t<T>[n]());

template<class T, class... Args> unspecified make_unique(Args&&...) = delete;

Constraints: T is an array of known bound.
template<class T> unique_ptr<T> make_unique_for_overwrite();

Constraints: T is not an array type.

Returns: unique_ptr<T>(new T).

template<class T> unique_ptr<T> make_unique_for_overwrite(size_t n);

Constraints: T is an array of unknown bound.

Returns: unique_ptr<T>(new remove_extent_t<T>[n]).

template<class T, class... Args> unspecified make_unique_for_overwrite(Args&&...)= delete;

Constraints: T is an array of known bound.

20.11.1.6 Specialized algorithms

template<class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;

Constraints: is_swappable_v<D> is true.

Effects: Calls x.swap(y).

template<class T1, class D1, class T2, class D2>
bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: x.get() == y.get().

Let CT denote

common_type_t<typename unique_ptr<T1, D1>::pointer,
    typename unique_ptr<T2, D2>::pointer>

Mandates:

(5.1) — unique_ptr<T1, D1>::pointer is implicitly convertible to CT and

(5.2) — unique_ptr<T2, D2>::pointer is implicitly convertible to CT.

Preconditions: The specialization less<CT> is a function object type (20.14) that induces a strict weak ordering (25.8) on the pointer values.

Returns: less<CT>()(x.get(), y.get()).

template<class T1, class D1, class T2, class D2>
bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: y < x.

template<class T1, class D1, class T2, class D2>
bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: !(y < x).

template<class T1, class D1, class T2, class D2>
requires three_way_comparable_with<typename unique_ptr<T1, D1>::pointer,
    typename unique_ptr<T2, D2>::pointer>

compare_three_way_result_t<typename unique_ptr<T1, D1>::pointer,
    typename unique_ptr<T2, D2>::pointer>

operator<=>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: compare_three_way()(x.get(), y.get()).

template<class T, class D>
bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;

Returns: !x.
template<class T, class D>
bool operator<(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
bool operator<(nullptr_t, const unique_ptr<T, D>& x);

Preconditions: The specialization less<unique_ptr<T, D>::pointer> is a function object type (20.14) that induces a strict weak ordering (25.8) on the pointer values.

Returns: The first function template returns less<unique_ptr<T, D>::pointer>()(x.get(), nullptr)
The second function template returns less<unique_ptr<T, D>::pointer>()(nullptr, x.get())

template<class T, class D>
bool operator>=(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
bool operator>=(nullptr_t, const unique_ptr<T, D>& x);

Returns: The first function template returns !nullptr < x. The second function template returns x < nullptr.

template<class T, class D>
bool operator>=(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
bool operator>=(nullptr_t, const unique_ptr<T, D>& x);

Returns: The first function template returns !(nullptr < x). The second function template returns !(x < nullptr).

template<class T, class D>

requires three_way_comparable_with<typename unique_ptr<T, D>::pointer, nullptr_t>

compare_three_way_result_t<typename unique_ptr<T, D>::pointer, nullptr_t>
operator<=>(const unique_ptr<T, D>& x, nullptr_t);

Returns: compare_three_way()(x.get(), nullptr).

20.11.1.7 I/O [unique.ptr.io]

template<class E, class T, class Y, class D>

basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const unique_ptr<Y, D>& p);

Constraints: os << p.get() is a valid expression.

Effects: Equivalent to: os << p.get();

Returns: os.

20.11.2 Class bad_weak_ptr [util.smartptr.weak.bad]

namespace std {

class bad_weak_ptr : public exception {

public:

   // see 17.9.3 for the specification of the special member functions
   const char* what() const noexcept override;

};

}

An exception of type bad_weak_ptr is thrown by the shared_ptr constructor taking a weak_ptr.

const char* what() const noexcept override;

Returns: An implementation-defined ntbs.
20.11.3  Class template shared_ptr

20.11.3.1  General

The shared_ptr class template stores a pointer, usually obtained via new. shared_ptr implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A shared_ptr is said to be empty if it does not own a pointer.

namespace std {
  template<class T> class shared_ptr {
    public:
      using element_type = remove_extent_t<T>;
      using weak_type = weak_ptr<T>;

      // 20.11.3.2, constructors
      constexpr shared_ptr() noexcept;
      constexpr shared_ptr(nullptr_t) noexcept : shared_ptr() { }
      template<class Y>
        explicit shared_ptr(Y* p);
      template<class Y, class D>
        shared_ptr(Y* p, D d);
      template<class Y, class D, class A>
        shared_ptr(Y* p, D d, A a);
      template<class D>
        shared_ptr(nullptr_t p, D d);
      template<class D, class A>
        shared_ptr(nullptr_t p, D d, A a);
      template<class Y>
        shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;
      template<class Y>
        shared_ptr(shared_ptr<Y>&& r, element_type* p) noexcept;
      shared_ptr(const shared_ptr& r) noexcept;
      template<class Y>
        shared_ptr(const shared_ptr<Y>& r) noexcept;
      shared_ptr(shared_ptr&& r) noexcept;
      template<class Y>
        explicit shared_ptr(const weak_ptr<Y>& r);
      template<class Y, class D>
        shared_ptr(unique_ptr<Y, D>&& r);

      // 20.11.3.3, destructor
      ~shared_ptr();

      // 20.11.3.4, assignment
      shared_ptr& operator=(const shared_ptr& r) noexcept;
      template<class Y>
        shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
      shared_ptr& operator=(shared_ptr&& r) noexcept;
      template<class Y>
        shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
      template<class Y, class D>
        shared_ptr& operator=(unique_ptr<Y, D>&& r);

      // 20.11.3.5, modifiers
      void swap(shared_ptr& r) noexcept;
      void reset() noexcept;
      template<class Y>
        void reset(Y* p);
      template<class Y, class D>
        void reset(Y* p, D d);
      template<class Y, class D, class A>
        void reset(Y* p, D d, A a);
  }
}
Specializations of `shared_ptr` shall be `Cpp17CopyConstructible`, `Cpp17CopyAssignable`, and `Cpp17LessThanComparable`, allowing their use in standard containers. Specializations of `shared_ptr` shall be contextually convertible to `bool`, allowing their use in boolean expressions and declarations in conditions.

The template parameter T of `shared_ptr` may be an incomplete type.

[Note 1: T can be a function type. —end note]

For purposes of determining the presence of a data race, member functions shall access and modify only the `shared_ptr` and `weak_ptr` objects themselves and not objects they refer to. Changes in `use_count()` do not reflect modifications that can introduce data races.

For the purposes of subclause 20.11, a pointer type Y* is said to be compatible with a pointer type T* when either Y* is convertible to T* or Y is U[N] and T is cv U[].

### 20.11.3.2 Constructors

In the constructor definitions below, enables `shared_from_this` with p, for a pointer p of type Y*, means that if Y has an unambiguous and accessible base class that is a specialization of `enable_shared_from_this` (20.11.6), then remove_cv_t<T>* shall be implicitly convertible to T* and the constructor evaluates the statement:

```c++
if (p != nullptr && p->weak_this.expired()) {
  p->weak_this = shared_ptr<remove_cv_t<T>>(*this, const_cast<remove_cv_t<T>*>(p));
```

The assignment to the `weak_this` member is not atomic and conflicts with any potentially concurrent access to the same object (6.9.2).

```c++
constexpr shared_ptr() noexcept;
```

2 Mandates: Y is a complete type.

3 **Constraints:** When T is an array type, the expression `delete[] p` is well-formed and either T is U[N] and Y(*)[N] is convertible to T*, or T is U[] and Y(*)[] is convertible to T*. When T is not an array type, the expression `delete p` is well-formed and Y* is convertible to T*.

4 **Preconditions:** The expression `delete[] p`, when T is an array type, or `delete p`, when T is not an array type, has well-defined behavior, and does not throw exceptions.

5 **Effects:** When T is not an array type, constructs a `shared_ptr` object that owns the pointer p. Otherwise, constructs a `shared_ptr` that owns p and a deleter of an unspecified type that calls `delete[] p`. When
T is not an array type, enables `shared_from_this` with p. If an exception is thrown, `delete` p is called when T is not an array type, `delete[]` p otherwise.

**Postconditions:** `use_count() == 1 && get() == p`.

**Throws:** `bad_alloc` or an implementation-defined exception when a resource other than memory could not be obtained.

```cpp
template<class Y, class D> shared_ptr(Y* p, D d);
template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
template<class D> shared_ptr(nullptr_t p, D d);
template<class D, class A> shared_ptr(nullptr_t p, D d, A a);
```

**Constraints:** `is_move_constructible_v<D>` is true, and `d(p)` is a well-formed expression. For the first two overloads:

- (9.1) If T is an array type, then either T is U[N] and Y(*)[N] is convertible to T*, or T is U[] and Y(*)[] is convertible to T*.
- (9.2) If T is not an array type, then Y* is convertible to T*.

**Preconditions:** Construction of d and a deleter of type D initialized with `std::move(d)` do not throw exceptions. The expression `d(p)` has well-defined behavior and does not throw exceptions. A meets the `Cpp17Allocator` requirements (Table 36).

**Effects:** Constructs a `shared_ptr` object that owns the object p and the deleter d. When T is not an array type, the first and second constructors enable `shared_from_this` with p. The second and fourth constructors shall use a copy of a to allocate memory for internal use. If an exception is thrown, `d(p)` is called.

**Postconditions:** `use_count() == 1 && get() == p`.

**Throws:** `bad_alloc` or an implementation-defined exception when a resource other than memory could not be obtained.

```cpp
template<class Y> shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;
template<class Y> shared_ptr(shared_ptr<Y>&& r, element_type* p) noexcept;
```

**Effects:** Constructs a `shared_ptr` instance that stores p and shares ownership with the initial value of r.

**Postconditions:** `get() == p`. For the second overload, r is empty and `r.get() == nullptr`.

[Note 1: Use of this constructor leads to a dangling pointer unless p remains valid at least until the ownership group of r is destroyed. —end note]

[Note 2: This constructor allows creation of an empty `shared_ptr` instance with a non-null stored pointer. —end note]

```cpp
shared_ptr(const shared_ptr& r) noexcept;
template<class Y> shared_ptr(const shared_ptr<Y>& r) noexcept;
```

**Constraints:** For the second constructor, Y* is compatible with T*.

**Effects:** If r is empty, constructs an empty `shared_ptr` object; otherwise, constructs a `shared_ptr` object that shares ownership with r.

**Postconditions:** `get() == r.get() && use_count() == r.use_count()`.

```cpp
template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
```

**Constraints:** Y* is compatible with T*.

**Effects:** Constructs a `shared_ptr` object that shares ownership with r and stores a copy of the pointer stored in r. If an exception is thrown, the constructor has no effect.

§ 20.11.3.2
Postconditions: use_count() == r.use_count().

Throws: bad_weak_ptr when r.expired().

template<class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);

Constraints: Y* is compatible with T* and unique_ptr<Y, D>::pointer is convertible to element_type*.

Effects: If r.get() == nullptr, equivalent to shared_ptr(). Otherwise, if D is not a reference type, equivalent to shared_ptr(r.release(), r.get_deleter()). Otherwise, equivalent to shared_ptr(r.release(), ref(r.get_deleter())). If an exception is thrown, the constructor has no effect.

20.11.3.3 Destructor [util.smartptr.shared.dest]

~shared_ptr();

Effects:

(1.1) — If *this is empty or shares ownership with another shared_ptr instance (use_count() > 1), there are no side effects.

(1.2) — Otherwise, if *this owns an object p and a deleter d, d(p) is called.

(1.3) — Otherwise, *this owns a pointer p, and delete p is called.

[Note 1: Since the destruction of *this decreases the number of instances that share ownership with *this by one, after *this has been destroyed all shared_ptr instances that shared ownership with *this will report a use_count() that is one less than its previous value. — end note]

20.11.3.4 Assignment [util.smartptr.shared.assign]

shared_ptr& operator=(const shared_ptr& r) noexcept;
template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;

Effects: Equivalent to shared_ptr(r).swap(*this).

Returns: *this.

[Note 1: The use count updates caused by the temporary object construction and destruction are not observable side effects, so the implementation can meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:

shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
both assignments can be no-ops. — end note]

shared_ptr& operator=(shared_ptr&& r) noexcept;
template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this.

template<class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this.

20.11.3.5 Modifiers [util.smartptr.shared.mod]

void swap(shared_ptr& r) noexcept;

Effects: Exchanges the contents of *this and r.

void reset() noexcept;

Effects: Equivalent to shared_ptr().swap(*this).

template<class Y> void reset(Y* p);

Effects: Equivalent to shared_ptr(p).swap(*this).
template<class Y, class D> void reset(Y* p, D d);
  *Effects: Equivalent to shared_ptr(p, d).swap(*this).*

template<class Y, class D, class A> void reset(Y* p, D d, A a);
  *Effects: Equivalent to shared_ptr(p, d, a).swap(*this).*

20.11.3.6 Observers

  element_type* get() const noexcept;
  *Returns: The stored pointer.*

  T& operator*() const noexcept;
  *Preconditions: get() != 0.*
  *Returns: *get.*
  *Remarks: When T is an array type or cv void, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

  T* operator->() const noexcept;
  *Preconditions: get() != 0.*
  *Returns: get.*
  *Remarks: When T is an array type, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

  element_type& operator[](ptrdiff_t i) const;
  *Preconditions: get() != 0 && i >= 0. If T is U[N], i < N.*
  *Returns: get()[i].
  *Throws: Nothing.*
  *Remarks: When T is not an array type, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

  long use_count() const noexcept;
  *Returns: The number of shared_ptr objects, *this included, that share ownership with *this, or 0 when *this is empty.*
  *Synchronization: None.*

  [Note 1: get() == nullptr does not imply a specific return value of use_count(). — end note]

  [Note 2: weak_ptr<T>::lock() can affect the return value of use_count(). — end note]

  [Note 3: When multiple threads might affect the return value of use_count(), the result is approximate. In particular, use_count() == 1 does not imply that accesses through a previously destroyed shared_ptr have in any sense completed. — end note]

  explicit operator bool() const noexcept;
  *Returns: get() != 0.*

  template<class U> bool owner_before(const shared_ptr<U>& b) const noexcept;
  template<class U> bool owner_before(const weak_ptr<U>& b) const noexcept;
  *Returns: An unspecified value such that
    (18.1) x.owner_before(y) defines a strict weak ordering as defined in 25.8;
    (18.2) under the equivalence relation defined by owner_before, !a.owner_before(b) & & !b.owner_before(a), two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both empty.*
The common requirements that apply to all `make_shared`, `allocate_shared`, `make_shared_for_overwrite`, and `allocate_shared_for_overwrite` overloads, unless specified otherwise, are described below.

```cpp
template<class T, ...>
shared_ptr<T> make_shared(args);

template<class T, class A, ...>
shared_ptr<T> allocate_shared(const A& a, args);

template<class T, ...>
shared_ptr<T> make_shared_for_overwrite(args);

template<class T, class A, ...>
shared_ptr<T> allocate_shared_for_overwrite(const A& a, args);
```

Preconditions: `A` meets the `Cpp17Allocator` requirements (Table 36).

Effects: Allocates memory for an object of type `T` (or `U[N]` when `T` is `U[]`, where `N` is determined from `args` as specified by the concrete overload). The object is initialized from `args` as specified by the concrete overload. The `allocate_shared` and `allocate_shared_for_overwrite` templates use a copy of `a` (rebound for an unspecified `value_type`) to allocate memory. If an exception is thrown, the functions have no effect.

Postconditions: `r.get() != 0 && r.use_count() == 1`, where `r` is the return value.

Returns: A `shared_ptr` instance that stores and owns the address of the newly constructed object.

Throws: `bad_alloc`, or an exception thrown from `allocate` or from the initialization of the object.

Remarks:

1. Implementations should perform no more than one memory allocation.

   [Note 1: This provides efficiency equivalent to an intrusive smart pointer. — end note]

2. When an object of an array type `U` is specified to have an initial value of `u` (of the same type), this shall be interpreted to mean that each array element of the object has as its initial value the corresponding element from `u`.

3. When an object of an array type is specified to have a default initial value, this shall be interpreted to mean that each array element of the object has a default initial value.

4. When a (sub)object of a non-array type `U` is specified to have an initial value of `v`, or `U(1...)`, where `1...` is a list of constructor arguments, `make_shared` shall initialize this (sub)object via the expression `::new(pv) U(v)` or `::new(pv) U(1...)` respectively, where `pv` has type `void*` and points to storage suitable to hold an object of type `U`.

5. When a (sub)object of a non-array type `U` is specified to have an initial value of `v`, or `U(1...)`, where `1...` is a list of constructor arguments, `allocate_shared` shall initialize this (sub)object via the expression

   ```cpp
   allocator_traits<A2>::construct(a2, pv, v) or
   allocator_traits<A2>::construct(a2, pv, 1...)
   ```

   respectively, where `pv` points to storage suitable to hold an object of type `U` and `a2` of type `A2` is a rebound copy of the allocator `a` passed to `allocate_shared` such that its `value_type` is `remove_cv_t<U>`.

6. When a (sub)object of non-array type `U` is specified to have a default initial value, `make_shared` shall initialize this (sub)object via the expression `::new(pv) U()`, where `pv` has type `void*` and points to storage suitable to hold an object of type `U`.

7. When a (sub)object of non-array type `U` is specified to have a default initial value, `allocate_shared` shall initialize this (sub)object via the expression `allocator_traits<A2>::construct(a2, pv)`, where `pv` points to storage suitable to hold an object of type `U` and `a2` of type `A2` is a rebound copy of the allocator `a` passed to `allocate_shared` such that its `value_type` is `remove_cv_t<U>`.

8. When a (sub)object of non-array type `U` is initialized by `make_shared_for_overwrite` or `allocate_shared_for_overwrite`, it is initialized via the expression `::new(pv) U`, where `pv` has type `void*` and points to storage suitable to hold an object of type `U`.

9. Array elements are initialized in ascending order of their addresses.
— When the lifetime of the object managed by the return value ends, or when the initialization of an array element throws an exception, the initialized elements are destroyed in the reverse order of their original construction.

— When a (sub)object of non-array type \( U \) that was initialized by \texttt{make\_shared} is to be destroyed, it is destroyed via the expression \( pv->U() \) where \( pv \) points to that object of type \( U \).

— When a (sub)object of non-array type \( U \) that was initialized by \texttt{allocate\_shared} is to be destroyed, it is destroyed via the expression \texttt{allocator\_traits\langle A2\rangle::destroy(a2, pv)} where \( pv \) points to that object of type \texttt{remove\_cv\_t\langle U \rangle} and \( a2 \) of type \( A2 \) is a rebound copy of the allocator \( a \) passed to \texttt{allocate\_shared} such that its \texttt{value\_type} is \texttt{remove\_cv\_t\langle U \rangle}.

[Note 2: These functions will typically allocate more memory than \texttt{sizeof(T)} to allow for internal bookkeeping structures such as reference counts. — end note]

\[
\begin{align*}
\text{template\textless \textit{class T}, \textit{class... Args}\textgreater \\
\text{shared\_ptr\langle T\rangle make\_shared(Args&&... args);} & \quad \text{\textemdash \textit{T} is not array} \\
\text{template\textless \textit{class T}, \textit{class A}, \textit{class... Args}\textgreater \\
\text{shared\_ptr\langle T\rangle allocate\_shared(const A& a, Args&&... args);} & \quad \text{\textit{T} is not array}
\end{align*}
\]

\textit{Constraints:} \( T \) is not an array type.

\textit{Returns:} A \textit{shared\_ptr} to an object of type \( T \) with an initial value \texttt{T(forward\langle Args\rangle(args)...)}.

\textit{Remarks:} The \textit{shared\_ptr} constructors called by these functions enable \textit{shared\_from\_this} with the address of the newly constructed object of type \( T \).

\textit{Example 1:}

\begin{align*}
\text{shared\_ptr\langle int\rangle p = make\_shared\langle int\rangle();} & \quad \text{\textit{shared\_ptr to int()} } \\
\text{shared\_ptr\langle vector\langle int\rangle\rangle q = make\_shared\langle vector\langle int\rangle\rangle(16, 1);} & \quad \text{\textit{shared\_ptr to vector of 16 elements with value 1}}
\end{align*}

— end example]

\texttt{template\textless \textit{class T}\rangle shared\_ptr\langle T\rangle make\_shared(size_t N);} & \quad \text{\textit{T} is \texttt{U[]} }
\texttt{template\textless \textit{class T}, \textit{class A}\textgt; \\
\text{shared\_ptr\langle T\rangle allocate\_shared(const A& a, size_t N);} & \quad \text{\textit{T} is \texttt{U[]} }
\]

\textit{Constraints:} \( T \) is of the form \texttt{U[]}.

\textit{Returns:} A \textit{shared\_ptr} to an object of type \texttt{U[N]} with a default initial value, where \( U \) is \texttt{remove\_extent\_t\langle T\rangle}.

\textit{Example 2:}

\begin{align*}
\text{shared\_ptr\langle double\rangle p = make\_shared\langle double\rangle(1024);} & \quad \text{\textit{shared\_ptr to a value-initialized double[1024]}} \\
\text{shared\_ptr\langle double\rangle[2][2] q = make\_shared\langle double\rangle[2][2](6);} & \quad \text{\textit{shared\_ptr to a value-initialized double[6][2][2]}}
\end{align*}

— end example]

\texttt{template\textless \textit{class T}\rangle \\
\text{shared\_ptr\langle T\rangle make\_shared();} & \quad \text{\textit{T} is \texttt{U[]} }
\texttt{template\textless \textit{class T}, \textit{class A}\textgt; \\
\text{shared\_ptr\langle T\rangle allocate\_shared(const A& a);} & \quad \text{\textit{T} is \texttt{U[]} }
\]

\textit{Constraints:} \( T \) is of the form \texttt{U[]}.

\textit{Returns:} A \textit{shared\_ptr} to an object of type \( T \) with a default initial value.

\textit{Example 3:}

\begin{align*}
\text{shared\_ptr\langle double[1024]\rangle p = make\_shared\langle double[1024]\rangle();} & \quad \text{\textit{shared\_ptr to a value-initialized double[1024]}} \\
\text{shared\_ptr\langle double[6][2][2]\rangle q = make\_shared\langle double[6][2][2]\rangle();} & \quad \text{\textit{shared\_ptr to a value-initialized double[6][2][2]}}
\end{align*}

— end example]

\texttt{template\textless \textit{class T}\rangle \\
\text{shared\_ptr\langle T\rangle make\_shared(size_t N, \quad \text{\textit{T} is \texttt{U[]}}}
\text{\texttt{\quad const remove\_extent\_t\langle T\rangle & u});} & \quad \text{\textit{T} is \texttt{U[]} }
\]
template<class T, class A>
shared_ptr<T> allocate_shared(const A & a, size_t N, const remove_extent_t<T> & u); // T is U[N]

Constraints: T is of the form U[].

Returns: A shared_ptr to an object of type U[N], where U is remove_extent_t<T> and each array element has an initial value of u.

[Example 4:
  shared_ptr<double[]> p = make_shared<double[]>(1024, 1.0);
    // shared_ptr to a double[1024], where each element is 1.0
  shared_ptr<double[6][2]> q = make_shared<double[6][2]>({1.0, 0.0});
    // shared_ptr to a double[6][2], where each double[2] element is {1.0, 0.0}
    // shared_ptr to a vector<int>[4], where each vector has contents {1, 2}
  —end example]

template<class T>
shared_ptr<T> make_shared(const remove_extent_t<T> & u); // T is U[N]
template<class T, class A>
shared_ptr<T> allocate_shared(const A & a, const remove_extent_t<T> & u); // T is U[N]

Constraints: T is of the form U[N].

Returns: A shared_ptr to an object of type T, where each array element of type remove_extent_t<T> has an initial value of u.

[Example 5:
  shared_ptr<double[1024]> p = make_shared<double[1024]>({1.0});
    // shared_ptr to a double[1024], where each element is 1.0
  shared_ptr<double[6][2]>& q = make_shared<double[6][2]>({1.0, 0.0});
    // shared_ptr to a double[6][2], where each double[2] element is {1.0, 0.0}
    // shared_ptr to a vector<int>[4], where each vector has contents {1, 2}
  —end example]

template<class T>
shared_ptr<T> make_shared_for_overwrite();
template<class T, class A>
shared_ptr<T> allocate_shared_for_overwrite(const A & a);

Constraints: T is not an array of unknown bound.

Returns: A shared_ptr to an object of type T.

[Example 6:
  struct X { double data[1024]; };
  shared_ptr<X> p = make_shared_for_overwrite<X>();
    // shared_ptr to a default-initialized X, where each element in X::data has an indeterminate value
  shared_ptr<double[1024]> q = make_shared_for_overwrite<double[1024]>();
    // shared_ptr to a default-initialized double[1024], where each element has an indeterminate value
  —end example]

template<class T>
shared_ptr<T> make_shared_for_overwrite(size_t N);
template<class T, class A>
shared_ptr<T> allocate_shared_for_overwrite(const A & a, size_t N);

Constraints: T is an array of unknown bound.

Returns: A shared_ptr to an object of type U[N], where U is remove_extent_t<T>.

[Example 7:
  shared_ptr<double[]> p = make_shared_for_overwrite<double[]>(1024);
    // shared_ptr to a default-initialized double[1024], where each element has an indeterminate value
  —end example]
20.11.3.8 Comparison

template<class T, class U>
bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
\textbf{Returns:} \texttt{a.get() == b.get}.

template<class T>
bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
\textbf{Returns:} \texttt{!a}.

template<class T, class U>
strong_ordering operator<=>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
\textbf{Returns:} \texttt{compare_three_way(a.get(), b.get())}.
\begin{note}
Defining a comparison operator function allows \texttt{shared_ptr} objects to be used as keys in associative containers. \end{note}

template<class T>
strong_ordering operator<=>(const shared_ptr<T>& a, nullptr_t) noexcept;
\textbf{Returns:} \texttt{compare_three_way(a.get(), nullptr)}.

20.11.3.9 Specialized algorithms

template<class T>
void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;
\textbf{Effects:} Equivalent to \texttt{a.swap(b)}.

20.11.3.10 Casts

template<class T, class U>
shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
shared_ptr<T> static_pointer_cast(shared_ptr<U>&& r) noexcept;
\begin{mandate}
The expression \texttt{static_cast<T*>((U*)\texttt{nullptr})} is well-formed.
\end{mandate}

\textbf{Returns:}
\begin{verbatim}
shared_ptr<T>(R, \texttt{static_cast<typename shared_ptr<T>::element_type*>(r.get())})
\end{verbatim}
where \texttt{R} is \texttt{r} for the first overload, and \texttt{std::move(r)} for the second.
\begin{note}
The seemingly equivalent expression \texttt{shared_ptr<T>\texttt{shared_ptr<T>::element_type*>(r.get())}} will eventually result in undefined behavior, attempting to delete the same object twice. \end{note}

template<class T, class U>
shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
shared_ptr<T> dynamic_pointer_cast(shared_ptr<U>&& r) noexcept;
\begin{mandate}
The expression \texttt{dynamic_cast<T*>(U*)\texttt{nullptr})} is well-formed. The expression \texttt{dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())} is well-formed.
\end{mandate}

\textbf{Preconditions:} The expression \texttt{dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())} has well-defined behavior.

\textbf{Returns:}
\begin{enumerate}
\item [6.1] When \texttt{dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())} returns a non-null value \texttt{p}, \texttt{shared_ptr<T>(R, p)}}, where \texttt{R} is \texttt{r} for the first overload, and \texttt{std::move(r)} for the second.
\item [6.2] Otherwise, \texttt{shared_ptr<T>\texttt{}}.
\end{enumerate}
\begin{note}
The seemingly equivalent expression \texttt{shared_ptr<T>(dynamic_cast<T*>(r.get())}} will eventually result in undefined behavior, attempting to delete the same object twice. \end{note}
template<class T, class U>
    shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;

template<class T, class U>
    shared_ptr<T> const_pointer_cast(shared_ptr<U>&& r) noexcept;

Mandates: The expression const_cast<T*>(U*)nullptr) is well-formed.

Returns:
    shared_ptr<T>(R, const_cast<typename shared_ptr<T>::element_type*>(r.get()))
    where R is r for the first overload, and std::move(r) for the second.

[Note 3: The seemingly equivalent expression shared_ptr<T>(const_cast<T*>(R).get())) will eventually result in undefined behavior, attempting to delete the same object twice. — end note]

template<class T, class U>
    shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;

template<class T, class U>
    shared_ptr<T> reinterpret_pointer_cast(shared_ptr<U>&& r) noexcept;

Mandates: The expression reinterpret_cast<T*>(U*)nullptr) is well-formed.

Returns:
    shared_ptr<T>(R, reinterpret_cast<typename shared_ptr<T>::element_type*>(r.get()))
    where R is r for the first overload, and std::move(r) for the second.

[Note 4: The seemingly equivalent expression shared_ptr<T>(reinterpret_cast<T*>(R).get())) will eventually result in undefined behavior, attempting to delete the same object twice. — end note]

20.11.3.11 get_deleter

template<class D, class T>
    D* get_deleter(const shared_ptr<T>& p) noexcept;

Returns: If p owns a deleter d of type cv-unqualified D, returns addressof(d); otherwise returns nullptr. The returned pointer remains valid as long as there exists a shared_ptr instance that owns d.

[Note 1: It is unspecified whether the pointer remains valid longer than that. This can happen if the implementation doesn’t destroy the deleter until all weak_ptr instances that share ownership with p have been destroyed. — end note]

20.11.3.12 I/O

template<class E, class T, class Y>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const shared_ptr<Y>& p);

Effects: As if by: os << p.get();

Returns: os.

20.11.4 Class template weak_ptr

20.11.4.1 General

The weak_ptr class template stores a weak reference to an object that is already managed by a shared_ptr. To access the object, a weak_ptr can be converted to a shared_ptr using the member function lock.

namespace std {
    template<class T>
        class weak_ptr {
            public:
                using element_type = remove_extent_t<T>;

                // 20.11.4.2, constructors
                constexpr weak_ptr() noexcept;
                template<class Y>
                    weak_ptr(const shared_ptr<Y>& r) noexcept;
                template<class Y>
                    weak_ptr(const weak_ptr<Y>& r) noexcept;
                template<class Y>
                    weak_ptr(weak_ptr<Y>&& r) noexcept;
                weak_ptr(weak_ptr<Y>&& r) noexcept;
            }

§ 20.11.4.1 668
Specializations of `weak_ptr` shall be `Cpp17CopyConstructible` and `Cpp17CopyAssignable`, allowing their use in standard containers. The template parameter `T` of `weak_ptr` may be an incomplete type.

### 20.11.4.2 Constructors

```cpp
template<class Y>
weak_ptr(weak_ptr<Y>&& r) noexcept;
```

// 20.11.4.3, destructor
-weak_ptr();

// 20.11.4.4, assignment
```cpp
weak_ptr& operator=(const weak_ptr& r) noexcept;
template<class Y>
weak_ptr& operator=(const weak_ptr<Y>& r) noexcept;
template<class Y>
weak_ptr& operator=(const shared_ptr<Y>& r) noexcept;
weak_ptr& operator=(weak_ptr& r) noexcept;
template<class Y>
weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;
```

// 20.11.4.5, modifiers
```cpp
void swap(weak_ptr& r) noexcept;
void reset() noexcept;
```

// 20.11.4.6, observers
```cpp
long use_count() const noexcept;
bool expired() const noexcept;
shared_ptr<T> lock() const noexcept;
template<class U>
bool owner_before(const shared_ptr<U>& b) const noexcept;
template<class U>
bool owner_before(const weak_ptr<U>& b) const noexcept;
};
```

```cpp
template<class T>
weak_ptr(shared_ptr<T>) -> weak_ptr<T>;
```

// 20.11.4.7, specialized algorithms
```cpp
template<class T>
void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
```
20.11.4.3 Destructor

\~weak_ptr();

Effects: Destroys this weak_ptr object but has no effect on the object its stored pointer points to.

20.11.4.4 Assignment

weak_ptr& operator=(const weak_ptr& r) noexcept;

Effects: Equivalent to weak_ptr(r).swap(*this).

Returns: *this.

Remarks: The implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary object.

weak_ptr& operator=(weak_ptr&& r) noexcept;

Effects: Equivalent to weak_ptr(std::move(r)).swap(*this).

Returns: *this.

20.11.4.5 Modifiers

void swap(weak_ptr& r) noexcept;

Effects: Exchanges the contents of *this and r.

Returns: *this.

20.11.4.6 Observers

long use_count() const noexcept;

Returns: 0 if *this is empty; otherwise, the number of shared_ptr instances that share ownership with *this.

bool expired() const noexcept;

Returns: use_count() == 0.

shared_ptr<T> lock() const noexcept;

Returns: expired() ? shared_ptr<T>() : shared_ptr<T>(*this), executed atomically.

template<class U> bool owner_before(const shared_ptr<U>& b) const noexcept;

template<class U> bool owner_before(const weak_ptr<U>& b) const noexcept;

Returns: An unspecified value such that

(4.1) x.owner_before(y) defines a strict weak ordering as defined in 25.8;

(4.2) under the equivalence relation defined by owner_before, !a.owner_before(b) && !b.owner_before(a), two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both empty.

20.11.4.7 Specialized algorithms

template<class T>
void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

Effects: Equivalent to a.swap(b).

20.11.5 Class template owner_less

The class template owner_less allows ownership-based mixed comparisons of shared and weak pointers.

namespace std {
    template<class T = void> struct owner_less;
}
template<class T> struct owner_less<shared_ptr<T>> {
    bool operator()(const shared_ptr<T>&, const shared_ptr<T>&) const noexcept;
    bool operator()(const shared_ptr<T>&, const weak_ptr<T>&) const noexcept;
    bool operator()(const weak_ptr<T>&, const shared_ptr<T>&) const noexcept;
};

template<class T> struct owner_less<weak_ptr<T>> {
    bool operator()(const weak_ptr<T>&, const weak_ptr<T>&) const noexcept;
    bool operator()(const shared_ptr<T>&, const weak_ptr<T>&) const noexcept;
    bool operator()(const weak_ptr<T>&, const shared_ptr<T>&) const noexcept;
};

template<> struct owner_less<void> {
    template<class T, class U>
    bool operator()(const shared_ptr<T>&, const shared_ptr<U>&) const noexcept;
    template<class T, class U>
    bool operator()(const shared_ptr<T>&, const weak_ptr<U>&) const noexcept;
    template<class T, class U>
    bool operator()(const weak_ptr<T>&, const shared_ptr<U>&) const noexcept;
    template<class T, class U>
    bool operator()(const weak_ptr<T>&, const weak_ptr<U>&) const noexcept;
    using is_transparent = unspecified;
};

operator()(x, y) returns x.owner_before(y).

[Note 1: Note that
— operator() defines a strict weak ordering as defined in 25.8;
— two shared_ptr or weak_ptr instances are equivalent under the equivalence relation defined by operator(), !operator(a, b) && !operator(b, a), if and only if they share ownership or are both empty.
— end note]

20.11.6 Class template enable_shared_from_this [util.smartptr.enab]

A class T can inherit from enable_shared_from_this<T> to inherit the shared_from_this member functions that obtain a shared_ptr instance pointing to *this.

[Example 1:

```cpp
struct X: public enable_shared_from_this<X> { }; 

int main() {
    shared_ptr<X> p(new X);
    shared_ptr<X> q = p->shared_from_this();
    assert(p == q);
    assert(!p.owner_before(q) && !q.owner_before(p)); // p and q share ownership
}

— end example]
```

namespace std {
    template<class T> class enable_shared_from_this {
    protected:
        constexpr enable_shared_from_this() noexcept;
        enable_shared_from_this(const enable_shared_from_this&) noexcept;
        enable_shared_from_this& operator=(const enable_shared_from_this&) noexcept;
        ~enable_shared_from_this();
    public:
        shared_ptr<T> shared_from_this();
        shared_ptr<T const> shared_from_this() const;
        weak_ptr<T> weak_from_this() noexcept;
        weak_ptr<T const> weak_from_this() const noexcept;
    };
private:
    mutable weak_ptr<T> weak_this;  // exposition only
};

The template parameter T of enable_shared_from_this may be an incomplete type.

constexpr enable_shared_from_this() noexcept;
enable_shared_from_this(const enable_shared_from_this<T>&) noexcept;

Effects: Value-initializes weak_this.

enable_shared_from_this<T>& operator=(const enable_shared_from_this<T>&) noexcept;

Returns: *this.

[Note 1: weak_this is not changed. — end note]

shared_ptr<T> shared_from_this();
shared_ptr<T const> shared_from_this() const;

Returns: shared_ptr<T>(weak_this).

weak_ptr<T> weak_from_this() noexcept;
weak_ptr<T const> weak_from_this() const noexcept;

Returns: weak_this.

20.11.7 Smart pointer hash support

template<class T, class D> struct hash<unique_ptr<T, D>>;

Letting UP be unique_ptr<T,D>, the specialization hash<UP> is enabled (20.14.19) if and only if
hash<typename UP::pointer> is enabled. When enabled, for an object p of type UP, hash<UP>()(p)
evaluates to the same value as hash<typename UP::pointer>()(p.get()). The member functions
are not guaranteed to be noexcept.

template<class T> struct hash<shared_ptr<T>>;

For an object p of type shared_ptr<T>, hash<shared_ptr<T>>()(p) evaluates to the same value as
hash<typename shared_ptr<T>::element_type*>()(p.get()).

20.12 Memory resources

20.12.1 Header <memory_resource> synopsis

namespace std::pmr {
    // 20.12.2, class memory_resource
    class memory_resource;

    bool operator==(const memory_resource& a, const memory_resource& b) noexcept;

    // 20.12.3, class template polymorphic_allocator
    template<class Tp = byte> class polymorphic_allocator;

    template<class T1, class T2>
    bool operator==(const polymorphic_allocator<T1>& a,
                   const polymorphic_allocator<T2>& b) noexcept;

    // 20.12.4, global memory resources
    memory_resource* new_delete_resource() noexcept;
    memory_resource* null_memory_resource() noexcept;
    memory_resource* set_default_resource(memory_resource* r) noexcept;
    memory_resource* get_default_resource() noexcept;

    // 20.12.5, pool resource classes
    struct pool_options;
    class synchronized_pool_resource;
    class unsynchronized_pool_resource;
class monotonic_buffer_resource;
}

20.12.2 Class memory_resource

20.12.2.1 General

The `memory_resource` class is an abstract interface to an unbounded set of classes encapsulating memory resources.

```cpp
namespace std::pmr {
    class memory_resource {
        static constexpr size_t max_align = alignof(max_align_t);  // exposition only

    public:
        memory_resource() = default;
        memory_resource(const memory_resource&) = default;
        virtual ~memory_resource();

        memory_resource& operator=(const memory_resource&) = default;

        [[nodiscard]] void* allocate(size_t bytes, size_t alignment = max_align);
        void deallocate(void* p, size_t bytes, size_t alignment = max_align);
        bool is_equal(const memory_resource& other) const noexcept;

    private:
        virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
        virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;
        virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
    };
}

20.12.2.2 Public member functions

~memory_resource();
    Effects: Destroys this `memory_resource`.

    [[nodiscard]] void* allocate(size_t bytes, size_t alignment = max_align);
    Effects: Equivalent to: return do_allocate(bytes, alignment);

    void deallocate(void* p, size_t bytes, size_t alignment = max_align);
    Effects: Equivalent to do_deallocate(p, bytes, alignment).

    bool is_equal(const memory_resource& other) const noexcept;
    Effects: Equivalent to: return do_is_equal(other);

20.12.2.3 Private virtual member functions

virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
    Preconditions: alignment is a power of two.
    Returns: A derived class shall implement this function to return a pointer to allocated storage (6.7.5.5.2) with a size of at least bytes, aligned to the specified alignment.
    Throws: A derived class implementation shall throw an appropriate exception if it is unable to allocate memory with the requested size and alignment.

virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;
    Preconditions: p was returned from a prior call to allocate(bytes, alignment) on a memory resource equal to *this, and the storage at p has not yet been deallocated.
    Effects: A derived class shall implement this function to dispose of allocated storage.
    Throws: Nothing.
```
virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;

7 Returns: A derived class shall implement this function to return true if memory allocated from this can be deallocated from other and vice-versa, otherwise false.

[Note 1: The most-derived type of other might not match the type of this. For a derived class D, an implementation of this function could immediately return false if dynamic_cast<const D*>(&other) == nullptr. — end note]

20.12.2.4 Equality [mem.res.eq]

bool operator==(const memory_resource& a, const memory_resource& b) noexcept;

1 Returns: a == b || a.is_equal(b).

20.12.3 Class template polymorphic_allocator [mem.poly.alloactor.class]

20.12.3.1 General [mem.poly.alloactor.class.general]

A specialization of class template pmr::polymorphic_allocator meets the Cpp17Allocator requirements (Table 36). Constructed with different memory resources, different instances of the same specialization of pmr::polymorphic_allocator can exhibit entirely different allocation behavior. This runtime polymorphism allows objects that use polymorphic_allocator to behave as if they used different allocator types at run time even though they use the same static allocator type.

2 All specializations of class template pmr::polymorphic_allocator meet the allocator completeness requirements (16.4.4.6.2).

namespace std::pmr {
// exposition only

template<class Tp = byte> class polymorphic_allocator {
    memory_resource* memory_rsrc;  // exposition only

public:
    using value_type = Tp;

    // 20.12.3.2, constructors
    polymorphic_allocator() noexcept;
    polymorphic_allocator(memory_resource* r);

    polymorphic_allocator(const polymorphic_allocator& other) = default;

    template<class U>
        polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;

    polymorphic_allocator& operator=(const polymorphic_allocator&) = delete;

    // 20.12.3.3, member functions
    [[nodiscard]] Tp* allocate(size_t n);
    void deallocate(Tp* p, size_t n);

    [[nodiscard]] void* allocate_bytes(size_t nbytes, size_t alignment = alignof(max_align_t));
    void deallocate_bytes(void* p, size_t nbytes, size_t alignment = alignof(max_align_t));

    template<class T> [[nodiscard]] T* allocate_object(size_t n = 1);
    template<class T> void deallocate_object(T* p, size_t n = 1);
    template<class T, class... CtorArgs> [[nodiscard]] T* new_object(CtorArgs&&... ctor_args);
    template<class T> void delete_object(T* p);

    template<class T, class... Args>
        void construct(T* p, Args&&... args);

    template<class T>
        void destroy(T* p);

    polymorphic_allocator select_on_container_copy_construction() const;

    memory_resource* resource() const;
};
20.12.3.2 Constructors

polymorphic_allocator() noexcept;
1  
   Effects: Sets memory_rsrc to get_default_resource().

polymorphic_allocator(memory_resource* r);
2  
   Preconditions: r is non-null.
   Effects: Sets memory_rsrc to r.
   Throws: Nothing.
   [Note 1: This constructor provides an implicit conversion from memory_resource*. — end note]

template<class U> polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;
3  
   Effects: Sets memory_rsrc to other.resource().

20.12.3.3 Member functions

[[nodiscard]] Tp* allocate(size_t n);
1  
   Effects: If numeric_limits<size_t>::max() / sizeof(Tp) < n, throws bad_array_new_length.
   Otherwise equivalent to:
   return static_cast<Tp*>(memory_rsrc->allocate(n * sizeof(Tp), alignof(Tp)));

void deallocate(Tp* p, size_t n);
2  
   Preconditions: p was allocated from a memory resource x, equal to *memory_rsrc, using x.allocate(n * sizeof(Tp), alignof(Tp)).
   Effects: Equivalent to memory_rsrc->deallocate(p, n * sizeof(Tp), alignof(Tp)).
   Throws: Nothing.

[[nodiscard]] void* allocate_bytes(size_t nbytes, size_t alignment = alignof(max_align_t));
5  
   Effects: Equivalent to:
   return memory_rsrc->allocate(nbytes, alignment);
   [Note 1: The return type is void* (rather than, e.g., byte*) to support conversion to an arbitrary pointer type U* by static_cast<U*>(p), thus facilitating construction of a U object in the allocated memory. — end note]

void deallocate_bytes(void* p, size_t nbytes, size_t alignment = alignof(max_align_t));
6  
   Effects: Equivalent to memory_rsrc->deallocate(p, nbytes, alignment).

template<class T>
[[nodiscard]] T* allocate_object(size_t n = 1);
8  
   Effects: Allocates memory suitable for holding an array of n objects of type T, as follows:
   (8.1) if numeric_limits<size_t>::max() / sizeof(T) < n, throws bad_array_new_length,
   (8.2) otherwise equivalent to:
   return static_cast<T*>(allocate_bytes(n*sizeof(T), alignof(T)));
   [Note 2: T is not deduced and must therefore be provided as a template argument. — end note]

template<class T>
void deallocate_object(T* p, size_t n = 1);
10  
   Effects: Equivalent to deallocate_bytes(p, n*sizeof(T), alignof(T)).

template<class T, class... CtorArgs>
[[nodiscard]] T* new_object(CtorArgs&&... ctor_args);
11  
   Effects: Allocates and constructs an object of type T, as follows.
   Equivalent to:
   T* p = allocate_object<T>();
   try {
      construct(p, std::forward<CtorArgs>(ctor_args)...);
   } catch (...) {
      deallocate_object(p);
throw;
}
return p;

[Note 3: T is not deduced and must therefore be provided as a template argument. — end note]

template<class T>
void delete_object(T* p);

Effects: Equivalent to:
destroy(p);
deallocate_object(p);

template<class T, class... Args>
void construct(T* p, Args&&... args);

Mandates: Uses-allocator construction of T with allocator *this (see 20.10.8.2) and constructor arguments std::forward<Args>(args)... is well-formed.

Effects: Construct a T object in the storage whose address is represented by p by uses-allocator construction with allocator *this and constructor arguments std::forward<Args>(args)....

Throws: Nothing unless the constructor for T throws.

template<class T>
void destroy(T* p);

Effects: As if by p->T().

polymorphic_allocator select_on_container_copy_construction() const;

Returns: polymorphic_allocator().

[Note 4: The memory resource is not propagated. — end note]

memory_resource* resource() const;

Returns: memory_rsrc.

20.12.3.4 Equality [mem.polyallocator.eq]

template<class T1, class T2>
bool operator==(const polymorphic_allocator<T1>& a,
const polymorphic_allocator<T2>& b) noexcept;

Returns: *a.resource() == *b.resource().

20.12.4 Access to program-wide memory_resource objects [mem.res.global]

memory_resource* new_delete_resource() noexcept;

Returns: A pointer to a static-duration object of a type derived from memory_resource that can serve as a resource for allocating memory using ::operator new and ::operator delete. The same value is returned every time this function is called. For a return value p and a memory resource r, p->is_equal(r) returns &r == p.

memory_resource* null_memory_resource() noexcept;

Returns: A pointer to a static-duration object of a type derived from memory_resource for which allocate() always throws bad_alloc and for which deallocate() has no effect. The same value is returned every time this function is called. For a return value p and a memory resource r, p->is_equal(r) returns &r == p.

The default memory resource pointer is a pointer to a memory resource that is used by certain facilities when an explicit memory resource is not supplied through the interface. Its initial value is the return value of new_delete_resource().

memory_resource* set_default_resource(memory_resource* r) noexcept;

Effects: If r is non-null, sets the value of the default memory resource pointer to r, otherwise sets the default memory resource pointer to new_delete_resource().

Returns: The previous value of the default memory resource pointer.
Remarks: Calling the `set_default_resource` and `get_default_resource` functions shall not incur a data race. A call to the `set_default_resource` function shall synchronize with subsequent calls to the `set_default_resource` and `get_default_resource` functions.

```cpp
memory_resource* get_default_resource() noexcept;
```

Returns: The current value of the default memory resource pointer.

## 20.12.5 Pool resource classes

### 20.12.5.1 Classes synchronized_pool_resource and unsynchronized_pool_resource

The `synchronized_pool_resource` and `unsynchronized_pool_resource` classes (collectively called `pool resource classes`) are general-purpose memory resources having the following qualities:

1. Each resource frees its allocated memory on destruction, even if `deallocate` has not been called for some of the allocated blocks.
2. A pool resource consists of a collection of `pools`, serving requests for different block sizes. Each individual pool manages a collection of `chunks` that are in turn divided into blocks of uniform size, returned via calls to `do_allocate`. Each call to `do_allocate(size, alignment)` is dispatched to the pool serving the smallest blocks accommodating at least `size` bytes.
3. When a particular pool is exhausted, allocating a block from that pool results in the allocation of an additional chunk of memory from the `upstream allocator` (supplied at construction), thus replenishing the pool. With each successive replenishment, the chunk size obtained increases geometrically.
   
   **Note 1:** By allocating memory in chunks, the pooling strategy increases the chance that consecutive allocations will be close together in memory. — end note
4. Allocation requests that exceed the largest block size of any pool are fulfilled directly from the upstream allocator.
5. A `pool_options` struct may be passed to the pool resource constructors to tune the largest block size and the maximum chunk size.

A `synchronized_pool_resource` may be accessed from multiple threads without external synchronization and may have thread-specific pools to reduce synchronization costs. An `unsynchronized_pool_resource` class may not be accessed from multiple threads simultaneously and thus avoids the cost of synchronization entirely in single-threaded applications.

```cpp
namespace std::pmr {
  struct pool_options {
    size_t max_blocks_per_chunk = 0;
    size_t largest_required_pool_block = 0;
  };

  class synchronized_pool_resource : public memory_resource {
    public:
      synchronized_pool_resource(const pool_options& opts, memory_resource* upstream);
      synchronized_pool_resource() :
          synchronized_pool_resource(pool_options(), get_default_resource()) {}
      explicit synchronized_pool_resource(memory_resource* upstream) :
          synchronized_pool_resource(pool_options(), upstream) {}
      explicit synchronized_pool_resource(const pool_options& opts) :
          synchronized_pool_resource(opts, get_default_resource()) {}
      synchronized_pool_resource(const synchronized_pool_resource&) = delete;
      virtual ~synchronized_pool_resource() = delete;

      void release();
      memory_resource* upstream_resource() const;
      pool_options options() const;

  };
```
protected:
    void* do_allocate(size_t bytes, size_t alignment) override;
    void do_deallocate(void* p, size_t bytes, size_t alignment) override;

    bool do_is_equal(const memory_resource& other) const noexcept override;
};

class unsynchronized_pool_resource : public memory_resource {
public:
    unsynchronized_pool_resource(const pool_options& opts, memory_resource* upstream);

    unsynchronized_pool_resource()
        : unsynchronized_pool_resource(pool_options(), get_default_resource()) {}
    explicit unsynchronized_pool_resource(memory_resource* upstream) { }
    explicit unsynchronized_pool_resource(const pool_options& opts) { }
    : unsynchronized_pool_resource(opts, get_default_resource()) {}

    unsynchronized_pool_resource(const unsynchronized_pool_resource&) = delete;
    virtual ~unsynchronized_pool_resource();

    unsynchronized_pool_resource& operator=(const unsynchronized_pool_resource&) = delete;

    void release();
    memory_resource* upstream_resource() const;
    pool_options options() const;

protected:
    void* do_allocate(size_t bytes, size_t alignment) override;
    void do_deallocate(void* p, size_t bytes, size_t alignment) override;

    bool do_is_equal(const memory_resource& other) const noexcept override;
};

20.12.5.2 pool_options data members

The members of pool_options comprise a set of constructor options for pool resources. The effect of each option on the pool resource behavior is described below:

size_t max_blocks_per_chunk;

The maximum number of blocks that will be allocated at once from the upstream memory resource (20.12.6) to replenish a pool. If the value of max_blocks_per_chunk is zero or is greater than an implementation-defined limit, that limit is used instead. The implementation may choose to use a smaller value than is specified in this field and may use different values for different pools.

size_t largest_required_pool_block;

The largest allocation size that is required to be fulfilled using the pooling mechanism. Attempts to allocate a single block larger than this threshold will be allocated directly from the upstream memory resource. If largest_required_pool_block is zero or is greater than an implementation-defined limit, that limit is used instead. The implementation may choose a pass-through threshold larger than specified in this field.

20.12.5.3 Constructors and destructors

synchronized_pool_resource(const pool_options& opts, memory_resource* upstream);
unsynchronized_pool_resource(const pool_options& opts, memory_resource* upstream);

Preconditions: upstream is the address of a valid memory resource.

Effects: Constructs a pool resource object that will obtain memory from upstream whenever the pool resource is unable to satisfy a memory request from its own internal data structures. The resulting object will hold a copy of upstream, but will not own the resource to which upstream points.
Note 1: The intention is that calls to upstream->allocate() will be substantially fewer than calls to this->allocate() in most cases. — end note

The behavior of the pooling mechanism is tuned according to the value of the opts argument.

Throws: Nothing unless upstream->allocate() throws. It is unspecified if, or under what conditions, this constructor calls upstream->allocate().

virtual ~synchronized_pool_resource();
virtual ~unsynchronized_pool_resource();

Effects: Calls release().

20.12.5.4 Members

void release();

Effects: Calls upstream_resource()->deallocate() as necessary to release all allocated memory.

Note 1: The memory is released back to upstream_resource() even if deallocate has not been called for some of the allocated blocks. — end note

Returns: The value of the upstream argument provided to the constructor of this object.

pool_options options() const;

Returns: The options that control the pooling behavior of this resource. The values in the returned struct may differ from those supplied to the pool resource constructor in that values of zero will be replaced with implementation-defined defaults, and sizes may be rounded to unspecified granularity.

void* do_allocate(size_t bytes, size_t alignment) override;

Effects: If the pool selected for a block of size bytes is unable to satisfy the memory request from its own internal data structures, it will call upstream_resource()->allocate() to obtain more memory. If bytes is larger than that which the largest pool can handle, then memory will be allocated using upstream_resource()->allocate().

Returns: A pointer to allocated storage (6.7.5.5.2) with a size of at least bytes. The size and alignment of the allocated memory shall meet the requirements for a class derived from memory_resource (20.12.2).

Throws: Nothing unless upstream_resource()->allocate() throws.

void do_deallocate(void* p, size_t bytes, size_t alignment) override;

Effects: Returns the memory at p to the pool. It is unspecified if, or under what circumstances, this operation will result in a call to upstream_resource()->deallocate().

Throws: Nothing.

bool do_is_equal(const memory_resource& other) const noexcept override;

Returns: this == &other.

20.12.6 Class monotonic_buffer_resource

20.12.6.1 General

A monotonic_buffer_resource is a special-purpose memory resource intended for very fast memory allocations in situations where memory is used to build up a few objects and then is released all at once when the memory resource object is destroyed. It has the following qualities:

(1.1) — A call to deallocate has no effect, thus the amount of memory consumed increases monotonically until the resource is destroyed.

(1.2) — The program can supply an initial buffer, which the allocator uses to satisfy memory requests.

(1.3) — When the initial buffer (if any) is exhausted, it obtains additional buffers from an upstream memory resource supplied at construction. Each additional buffer is larger than the previous one, following a geometric progression.

(1.4) — It is intended for access from one thread of control at a time. Specifically, calls to allocate and deallocate do not synchronize with one another.
— It frees the allocated memory on destruction, even if \texttt{deallocate} has not been called for some of the allocated blocks.

namespace std::pmr {

class monotonic_buffer_resource : public memory_resource {

memory_resource* upstream_rsrc; // exposition only
void* current_buffer; // exposition only
size_t next_buffer_size; // exposition only
}

public:

explicit monotonic_buffer_resource(memory_resource* upstream);

monotonic_buffer_resource(size_t initial_size, memory_resource* upstream);

monotonic_buffer_resource(void* buffer, size_t buffer_size, memory_resource* upstream);

monotonic_buffer_resource() : monotonic_buffer_resource(get_default_resource()) {} 

explicit monotonic_buffer_resource(size_t initial_size) : monotonic_buffer_resource(initial_size, get_default_resource()) {} 

monotonic_buffer_resource(void* buffer, size_t buffer_size) : monotonic_buffer_resource(buffer, buffer_size, get_default_resource()) {} 

monotonic_buffer_resource(const monotonic_buffer_resource&) = delete;

virtual ~monotonic_buffer_resource();

monotonic_buffer_resource& operator=(const monotonic_buffer_resource&) = delete;

void release();

memory_resource* upstream_resource() const;

protected:

void* do_allocate(size_t bytes, size_t alignment) override;

void do_deallocate(void* p, size_t bytes, size_t alignment) override;

bool do_is_equal(const memory_resource& other) const noexcept override;
};

20.12.6.2 Constructors and destructor

\textit{[mem.res.monotonic.buffer ctor]} 

\begin{flushleft}

\begin{enumerate}

\item explicit monotonic_buffer_resource(memory_resource* upstream);

\item monotonic_buffer_resource(size_t initial_size, memory_resource* upstream);

\begin{flushleft}

\begin{enumerate}

\item \textit{Preconditions}: upstream is the address of a valid memory resource. initial\_size, if specified, is greater than zero.

\item \textit{Effects}: Sets upstream\_rsrc to upstream and current\_buffer to nullptr. If initial\_size is specified, sets next\_buffer\_size to at least initial\_size; otherwise sets next\_buffer\_size to an implementation-defined size.

\item monotonic_buffer_resource(void* buffer, size_t buffer\_size, memory_resource* upstream);

\begin{flushleft}

\begin{enumerate}

\item \textit{Preconditions}: upstream is the address of a valid memory resource. buffer\_size is no larger than the number of bytes in buffer.

\item \textit{Effects}: Sets upstream\_rsrc to upstream, current\_buffer to buffer, and next\_buffer\_size to buffer\_size (but not less than 1), then increases next\_buffer\_size by an implementation-defined growth factor (which need not be integral).

\item \textit{~monotonic_buffer_resource();}

\item \textit{Effects}: Calls release().

\end{enumerate}

\end{flushleft}

\end{enumerate}

\end{flushleft}

\end{enumerate}

20.12.6.3 Members

\textit{[mem.res.monotonic.buffer mem]} 

\begin{flushleft}

void release();

\begin{enumerate}

\item \textit{Effects}: Calls upstream\_rsrc\texttt{\textasciitilde}deallocate() as necessary to release all allocated memory.

\end{enumerate}

\end{flushleft}
Note 1: The memory is released back to `upstream_rsrc` even if some blocks that were allocated from this have not been deallocated from this. — end note

memory_resource* upstream_resource() const;

Returns: The value of `upstream_rsrc`.

void* do_allocate(size_t bytes, size_t alignment) override;

Effects: If the unused space in current_buffer can fit a block with the specified bytes and alignment, then allocate the return block from current_buffer; otherwise set current_buffer to upstream_rsrc->allocate(n, m), where n is not less than max(bytes, next_buffer_size) and m is not less than alignment, and increase next_buffer_size by an implementation-defined growth factor (which need not be integral), then allocate the return block from the newly-allocated current_buffer.

Returns: A pointer to allocated storage (6.7.5.5.2) with a size of at least bytes. The size and alignment of the allocated memory shall meet the requirements for a class derived from memory_resource (20.12.2).

Throws: Nothing unless upstream_rsrc->allocate() throws.

void do_deallocate(void* p, size_t bytes, size_t alignment) override;

Effects: None.

Throws: Nothing.

Remarks: Memory used by this resource increases monotonically until its destruction.

bool do_is_equal(const memory_resource& other) const noexcept override;

Returns: this == &other.

20.13 Class template scoped_allocator_adaptor

namespace std {

// class template scoped allocator adaptor
template<class OuterAlloc, class... InnerAllocs>
class scoped_allocator_adaptor;

// 20.13.5, scoped allocator operators
template<class OuterA1, class OuterA2, class... InnerAllocs>
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
}

The class template `scoped_allocator_adaptor` is an allocator template that specifies an allocator resource (the outer allocator) to be used by a container (as any other allocator does) and also specifies an inner allocator resource to be passed to the constructor of every element within the container. This adaptor is instantiated with one outer and zero or more inner allocator types. If instantiated with only one allocator type, the inner allocator becomes the `scoped_allocator_adaptor` itself, thus using the same allocator resource for the container and every element within the container and, if the elements themselves are containers, each of their elements recursively. If instantiated with more than one allocator, the first allocator is the outer allocator for use by the container, the second allocator is passed to the constructors of the container’s elements, and, if the elements themselves are containers, the third allocator is passed to the elements’ elements, and so on. If containers are nested to a depth greater than the number of allocators, the last allocator is used repeatedly, as in the single-allocator case, for any remaining recursions.

Note 1: The `scoped_allocator_adaptor` is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions. — end note]
public:
using outer_allocator_type = OuterAlloc;
using inner_allocator_type = see below;

using value_type = typename OuterTraits::value_type;
using size_type = typename OuterTraits::size_type;
using difference_type = typename OuterTraits::difference_type;
using pointer = typename OuterTraits::pointer;
using const_pointer = typename OuterTraits::const_pointer;
using void_pointer = typename OuterTraits::void_pointer;
using const_void_pointer = typename OuterTraits::const_void_pointer;

using propagate_on_container_copy_assignment = see below;
using propagate_on_container_move_assignment = see below;
using propagate_on_container_swap = see below;
using is_always_equal = see below;

template<class Tp> struct rebind {
    using other = scoped_allocator_adaptor<
        OuterTraits::template rebind_alloc<Tp>, InnerAllocs...>;
};

scoped_allocator_adaptor();
template<class OuterA2>
    scoped_allocator_adaptor(OuterA2&& outerAlloc,
        const InnerAllocs&... innerAllocs) noexcept;

scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;
scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;

template<class OuterA2>
    scoped_allocator_adaptor(
        const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;

    template<class OuterA2>
    scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;

    scoped_allocator_adaptor& operator=(const scoped_allocator_adaptor&) = default;
    scoped_allocator_adaptor& operator=(scoped_allocator_adaptor&&) = default;

~scoped_allocator_adaptor();

inner_allocator_type& inner_allocator() noexcept;
const inner_allocator_type& inner_allocator() const noexcept;
outer_allocator_type& outer_allocator() noexcept;
const outer_allocator_type& outer_allocator() const noexcept;

[[nodiscard]] pointer allocate(size_type n);
[[nodiscard]] pointer allocate(size_type n, const_void_pointer hint);
void deallocate(pointer p, size_type n);
size_type max_size() const;

template<T, class... Args>
    void construct(T* p, Args&&... args);

template<T>
    void destroy(T* p);

scoped_allocator_adaptor select_on_container_copy_construction() const;
template<class OuterAlloc, class... InnerAllocs>
scoped_allocator_adaptor(OuterAlloc, InnerAllocs...) -> scoped_allocator_adaptor<OuterAlloc, InnerAllocs...>;

20.13.2 Member types

using inner_allocator_type = see below;  
Type: scoped_allocator_adaptor<OuterAlloc> if sizeof...(InnerAllocs) is zero; otherwise, scoped_allocator_adaptor<InnerAllocs...>.

using propagate_on_container_copy_assignment = see below;  
Type: true_type if allocator_traits<A>::propagate_on_container_copy_assignment::value is true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

using propagate_on_container_move_assignment = see below;  
Type: true_type if allocator_traits<A>::propagate_on_container_move_assignment::value is true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

using propagate_on_container_swap = see below;  
Type: true_type if allocator_traits<A>::propagate_on_container_swap::value is true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

using is_always_equal = see below;  
Type: true_type if allocator_traits<A>::is_always_equal::value is true for every A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

20.13.3 Constructors

scoped_allocator_adaptor();  
Effects: Value-initializes the OuterAlloc base class and the inner allocator object.

template<class OuterA2>
scoped_allocator_adaptor(OuterA2&& outerAlloc, const InnerAllocs&... innerAllocs) noexcept;  
Constraints: is_constructible_v<OuterAlloc, OuterA2> is true.  
Effects: Initializes the OuterAlloc base class with std::forward<OuterA2>(outerAlloc) and inner with innerAllocs... (hence recursively initializing each allocator within the adaptor with the corresponding allocator from the argument list).

scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;  
Effects: Initializes each allocator within the adaptor with the corresponding allocator from other.

scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;  
Effects: Move constructs each allocator within the adaptor with the corresponding allocator from other.

template<class OuterA2>
scoped_allocator_adaptor(
   const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;  
Constraints: is_constructible_v<OuterAlloc, OuterA2> is true.  
Effects: Initializes each allocator within the adaptor with the corresponding allocator from other.

template<class OuterA2>
scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;  
Constraints: is_constructible_v<OuterAlloc, OuterA2> is true.  
Effects: Initializes each allocator within the adaptor with the corresponding allocator rvalue from other.
20.13.4 Members

In the construct member functions, `OUTERMOST(x)` is `OUTERMOST(x.outer_allocator())` if the expression `x.outer_allocator()` is valid (13.10.3) and `x` otherwise; `OUTERMOST_ALLOC_TRAITS(x)` is `allocator_traits<remove_reference_t<decltype(OUTERMOST(x))>>`.

[Note 1: `OUTERMOST(x)` and `OUTERMOST_ALLOC_TRAITS(x)` are recursive operations. It is incumbent upon the definition of `outer_allocator()` to ensure that the recursion terminates. It will terminate for all instantiations of `scoped_allocator_adaptor`. — end note]

```c
inner_allocator_type& inner_allocator() noexcept;
const inner_allocator_type& inner_allocator() const noexcept;
```

Returns: `*this` if `sizeof...(InnerAllocs)` is zero; otherwise, `inner`.

```c
outer_allocator_type& outer_allocator() noexcept;
```

Returns: `static_cast<OuterAlloc&>(*this)`.

```c
const outer_allocator_type& outer_allocator() const noexcept;
```

Returns: `static_cast<const OuterAlloc&>(*this)`.

```c
[nodiscard] pointer allocate(size_type n);
```

Returns: `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n)`.

```c
[nodiscard] pointer allocate(size_type n, const_void_pointer hint);
```

Returns: `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n, hint)`.

```c
void deallocate(pointer p, size_type n) noexcept;
```

Effects: As if by: `allocator_traits<OuterAlloc>::deallocate(outer_allocator(), p, n)`.

```c
size_type max_size() const;
```

Returns: `allocator_traits<OuterAlloc>::max_size(outer_allocator())`.

```c
template<class T, class... Args>
void construct(T* p, Args&&... args);
```

Effects: Equivalent to:

```c
apply([p, this](auto&&... newargs) {
    OUTERMOST_ALLOC_TRAITS(*this)::construct(
        OUTERMOST(*this), p,
        std::forward<decltype(newargs)>(newargs)...);
},
    uses_allocator_construction_args<T>(inner_allocator(),
        std::forward<Args>(args)...));
```

```c
template<class T>
void destroy(T* p);
```

Effects: Calls `OUTERMOST_ALLOC_TRAITS(*this)::destroy(OUTERMOST(*this), p)`.

```c
scoped_allocator_adaptor select_on_container_copy_construction() const;
```

Returns: A new `scoped_allocator_adaptor` object where each allocator `A` in the adaptor is initialized from the result of calling `allocator_traits<A>::select_on_container_copy_construction()` on the corresponding allocator in `*this`.

20.13.5 Operators

```c
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

Returns: If `sizeof...(InnerAllocs)` is zero,

```c
    a.outer_allocator() == b.outer_allocator()
```

otherwise
a.outer_allocator() == b.outer_allocator() && a.inner_allocator() == b.inner_allocator()

20.14 Function objects

20.14.1 General

A function object type is an object type (6.8) that can be the type of the postfix-expression in a function call (7.6.1.3, 12.4.2.2). A function object is an object of a function object type. In the places where one would expect to pass a pointer to a function to an algorithmic template (Clause 25), the interface is specified to accept a function object. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

20.14.2 Header <functional> synopsis

namespace std {
  // 20.14.5, invoke
  template<class F, class... Args>
    constexpr invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
      noexcept(is_nothrow_invocable_v<F, Args...>);
  // 20.14.6, reference_wrapper
  template<class T> class reference_wrapper;
  template<class T> constexpr reference_wrapper<T> ref(T&) noexcept;
  template<class T> constexpr reference_wrapper<const T> cref(const T&) noexcept;
  template<class T> void ref(const T&) = delete;
  template<class T> void cref(const T&) = delete;
  template<class T> constexpr reference_wrapper<T> ref(reference_wrapper<T>) noexcept;
  template<class T> constexpr reference_wrapper<const T> cref(reference_wrapper<T>) noexcept;
  // 20.14.7, arithmetic operations
  template<class T = void> struct plus;
  template<class T = void> struct minus;
  template<class T = void> struct multiplies;
  template<class T = void> struct divides;
  template<class T = void> struct modulus;
  template<class T = void> struct negate;
  template<> struct plus<void>;
  template<> struct minus<void>;
  template<> struct multiplies<void>;
  template<> struct divides<void>;
  template<> struct modulus<void>;
  template<> struct negate<void>;
  // 20.14.8, comparisons
  template<class T = void> struct equal_to;
  template<class T = void> struct not_equal_to;
  template<class T = void> struct greater;
  template<class T = void> struct less;
  template<class T = void> struct greater_equal;
  template<class T = void> struct less_equal;
  template<> struct equal_to<void>;
  template<> struct not_equal_to<void>;
  template<> struct greater<void>;
  template<> struct less<void>;
  template<> struct greater_equal<void>;
  template<> struct less_equal<void>;
  // 20.14.8.8, class compare_three_way
  struct compare_three_way;
}

225) Such a type is a function pointer or a class type which has a member operator() or a class type which has a conversion to a pointer to function.
// 20.14.10, logical operations
template<class T = void> struct logical_and;
template<class T = void> struct logical_or;
template<class T = void> struct logical_not;
template<> struct logical_and<void>;
template<> struct logical_or<void>;
template<> struct logical_not<void>;

// 20.14.11, bitwise operations
template<class T = void> struct bit_and;
template<class T = void> struct bit_or;
template<class T = void> struct bit_xor;
template<class T = void> struct bit_not;
template<> struct bit_and<void>;
template<> struct bit_or<void>;
template<> struct bit_xor<void>;
template<> struct bit_not<void>;

// 20.14.12, identity
struct identity;

// 20.14.13, function template not_fn
template<class F> constexpr unspecified not_fn(F&& f);

// 20.14.14, function template bind_front
template<class F, class... Args> constexpr unspecified bind_front(F&&, Args&&...);

// 20.14.15, bind
template<class T> struct is_bind_expression;
template<class T>
inline constexpr bool is_bind_expression_v = is_bind_expression<T>::value;
template<class T>
inline constexpr int is_placeholder_v = is_placeholder<T>::value;
template<class F, class... BoundArgs>
constexpr unspecified bind(F&&, BoundArgs&&...);
template<class R, class F, class... BoundArgs>
constexpr unspecified bind(F&&, BoundArgs&&...);

namespace placeholders {
    // M is the implementation-defined number of placeholders
    see below _1;
    see below _2;
    
    see below _M;
}

// 20.14.16, member function adaptors
template<class R, class T>
constexpr unspecified mem_fn(R T::* noexcept;

// 20.14.17, polymorphic function wrappers
class bad_function_call;

template<class> class function;  // not defined
template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

template<class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;
template<class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

// 20.14.18, searchers
template<class ForwardIterator, class BinaryPredicate = equal_to<>>
    class default_searcher;

template<class RandomAccessIterator,
         class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
         class BinaryPredicate = equal_to<>>
    class boyer_moore_searcher;

template<class RandomAccessIterator,
         class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
         class BinaryPredicate = equal_to<>>
    class boyer_moore_horspool_searcher;

// 20.14.19, class template hash
template<class T>
    struct hash;

namespace ranges {
    // 20.14.9, concept-constrained comparisons
    struct equal_to;
    struct not_equal_to;
    struct greater;
    struct less;
    struct greater_equal;
    struct less_equal;
}

[Example 1: If a C++ program wants to have a by-element addition of two vectors a and b containing double and put
the result into a, it can do:
    transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
    —end example]

[Example 2: To negate every element of a:
    transform(a.begin(), a.end(), a.begin(), negate<double>());
    —end example]

20.14.3 Definitions

The following definitions apply to this Clause:

1 A call signature is the name of a return type followed by a parenthesized comma-separated list of zero or
   more argument types.

2 A callable type is a function object type (20.14) or a pointer to member.

3 A callable object is an object of a callable type.

4 A call wrapper type is a type that holds a callable object and supports a call operation that forwards to that
   object.

5 A call wrapper is an object of a call wrapper type.

6 A target object is the callable object held by a call wrapper.

7 A call wrapper type may additionally hold a sequence of objects and references that may be passed as
   arguments to the target object. These entities are collectively referred to as bound argument entities.

8 The target object and bound argument entities of the call wrapper are collectively referred to as state entities.

20.14.4 Requirements

1 Define INVOKE(f, t1, t2, ..., tN) as follows:
Every call wrapper (20.14.3) meets the Cpp17MoveConstructible and Cpp17Destructible requirements. An argument forwarding call wrapper is a call wrapper that can be called with an arbitrary argument list and delivers the arguments to the wrapped callable object as references. This forwarding step delivers rvalue arguments as rvalue references and lvalue arguments as lvalue references.

[Note 1: In a typical implementation, argument forwarding call wrappers have an overloaded function call operator of the form

```cpp
template<class... UnBoundArgs>
constexpr R operator()(UnBoundArgs&&... unbound_args) cv-qual;
```

— end note]

A perfect forwarding call wrapper is an argument forwarding call wrapper that forwards its state entities to the underlying call expression. This forwarding step delivers a state entity of type T as cv T& when the call is performed on an lvalue of the call wrapper type and as cv T&& otherwise, where cv represents the cv-qualifiers of the call wrapper and where cv shall be neither volatile nor const volatile.

A call pattern defines the semantics of invoking a perfect forwarding call wrapper. A postfix call performed on a perfect forwarding call wrapper is expression-equivalent (3.21) to an expression e determined from its call pattern cp by replacing all occurrences of the arguments of the call wrapper and its state entities with references as described in the corresponding forwarding steps.

A simple call wrapper is a perfect forwarding call wrapper that meets the Cpp17CopyConstructible and Cpp17CopyAssignable requirements and whose copy constructor, move constructor, and assignment operators are constexpr functions that do not throw exceptions.

The copy/move constructor of an argument forwarding call wrapper has the same apparent semantics as if memberwise copy/move of its state entities were performed (11.4.5.3).

[Note 2: This implies that each of the copy/move constructors has the same exception-specification as the corresponding implicit definition and is declared as constexpr if the corresponding implicit definition would be considered to be constexpr. — end note]

Argument forwarding call wrappers returned by a given standard library function template have the same type if the types of their corresponding state entities are the same.

20.14.5 Function template invoke

```cpp
template<class F, class... Args>
constexpr invoke_result_t<T, Args...> invoke(F&& f, Args&&... args)
noexcept(is_nothrow_invocable_v<F, Args...>);
```

Returns: INVOKE(std::forward<F>(f), std::forward<Args>(args)...) (20.14.4).

20.14.6 Class template reference_wrapper

20.14.6.1 General

```cpp
namespace std {
```
template<class T> class reference_wrapper {
public:
    // types
    using type = T;

    // construct/copy/destroy
    template<class U>
        constexpr reference_wrapper(U&&) noexcept((see below));
    constexpr reference_wrapper(const reference_wrapper& x) noexcept;

    // assignment
    constexpr reference_wrapper& operator=(const reference_wrapper& x) noexcept;

    // access
    constexpr operator T& () const noexcept;
    constexpr T& get() const noexcept;

    // invocation
    template<class... ArgTypes>
        constexpr invoke_result_t<T&, ArgTypes...> operator()(ArgTypes&&...) const;
};

template<class T>
reference_wrapper<T> -> reference_wrapper<T>;

1 reference_wrapper<T> is a Cpp17CopyConstructible and Cpp17CopyAssignable wrapper around a reference to an object or function of type T.

2 reference_wrapper<T> is a trivially copyable type (6.8).

3 The template parameter T of reference_wrapper may be an incomplete type.

20.14.6.2 Constructors and destructor [refwrap.const]

    template<class U>
        constexpr reference_wrapper(U&& u) noexcept((see below));

1 Let FUN denote the exposition-only functions
    void FUN(T&) noexcept;
    void FUN(T&&) = delete;

2 Constraints: The expression FUN(declval<U>()) is well-formed and is_same_v<remove_cvref_t<U>,
    reference_wrapper> is false.

3 Effects: Creates a variable r as if by T& r = std::forward<U>(u), then constructs a reference_-
    wrapper object that stores a reference to r.

4 Remarks: The expression inside noexcept is equivalent to noexcept(FUN(declval<U>())).

    constexpr reference_wrapper(const reference_wrapper& x) noexcept;

1 Effects: Constructs a reference_wrapper object that stores a reference to x.get().

20.14.6.3 Assignment [refwrap.assign]

    constexpr reference_wrapper& operator=(const reference_wrapper& x) noexcept;

1 Postconditions: *this stores a reference to x.get().

20.14.6.4 Access [refwrap.access]

    constexpr operator T& () const noexcept;

1 Returns: The stored reference.

    constexpr T& get() const noexcept;

2 Returns: The stored reference.
20.14.6.5 Invocation

```cpp
template<class... ArgTypes>
constexpr invoke_result_t<T&, ArgTypes...> operator()(ArgTypes&&... args) const;
```

1  **Mandates**: T is a complete type.
2  **Returns**: INVOKE(get(), std::forward<ArgTypes>(args)...). (20.14.4)

20.14.6.6 Helper functions

```cpp
template<class T> constexpr reference_wrapper<T> ref(T& t) noexcept;
```

1  **Returns**: reference_wrapper<T>(t).

```cpp
template<class T> constexpr reference_wrapper<T> ref(reference_wrapper<T> t) noexcept;
```

2  **Returns**: ref(t.get()).

```cpp
template<class T> constexpr reference_wrapper<const T> cref(const T& t) noexcept;
```

3  **Returns**: cref(t.get()).

```
```cpp
template<class T> constexpr reference_wrapper<const T> cref(reference_wrapper<T> t) noexcept;
```

4  **Returns**: cref(t.get()).

20.14.7 Arithmetic operations

20.14.7.1 General

1  The library provides basic function object classes for all of the arithmetic operators in the language (7.6.5, 7.6.6).

20.14.7.2 Class template plus

```cpp
template<class T = void> struct plus {
    constexpr T operator()(const T& x, const T& y) const;
    
    constexpr T operator()(const T& x, const T& y) const;
```

1  **Returns**: x + y.

```cpp
template<> struct plus<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) + std::forward<U>(u));
```

2  **Returns**: std::forward<T>(t) + std::forward<U>(u).
using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
  -> decltype(std::forward<T>(t) - std::forward<U>(u));

2 Returns: std::forward<T>(t) - std::forward<U>(u).

20.14.7.4 Class template multiplies
[arithmetic.operations.multiplies]
template<class T = void> struct multiplies {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
1 Returns: x * y.

template<> struct multiplies<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) * std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
  -> decltype(std::forward<T>(t) * std::forward<U>(u));
2 Returns: std::forward<T>(t) * std::forward<U>(u).

20.14.7.5 Class template divides
[arithmetic.operations.divides]
template<class T = void> struct divides {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
1 Returns: x / y.

template<> struct divides<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) / std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
  -> decltype(std::forward<T>(t) / std::forward<U>(u));
2 Returns: std::forward<T>(t) / std::forward<U>(u).

20.14.7.6 Class template modulus
[arithmetic.operations.modulus]
template<class T = void> struct modulus {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
1 Returns: x % y.

template<> struct modulus<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) % std::forward<U>(u));

    using is_transparent = unspecified;
};
template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) % std::forward<U>(u));

2

Returns: std::forward<T>(t) % std::forward<U>(u).

20.14.7.7 Class template negate

template<class T = void> struct negate {
    constexpr T operator()(const T& x) const;
};

constexpr T operator()(const T& x) const;

1

Returns: -x.

template<> struct negate<void> {
    template<class T> constexpr auto operator()(T&& t) const
        -> decltype(-std::forward<T>(t));
    using is_transparent = unspecified;
};

template<class T> constexpr auto operator()(T&& t) const
    -> decltype(-std::forward<T>(t));

2

Returns: -std::forward<T>(t).

20.14.8 Comparisons

20.14.8.1 General

The library provides basic function object classes for all of the comparison operators in the language (7.6.9, 7.6.10).

2 For templates less, greater, less_equal, and greater_equal, the specializations for any pointer type yield a result consistent with the implementation-defined strict total order over pointers (3.27).

[Note 1: If a < b is well-defined for pointers a and b of type P, then (a < b) == less<P>()(a, b), (a > b) == greater<P>()(a, b), and so forth. —end note]

For template specializations less<void>, greater<void>, less_equal<void>, and greater_equal<void>, if the call operator calls a built-in operator comparing pointers, the call operator yields a result consistent with the implementation-defined strict total order over pointers.

20.14.8.2 Class template equal_to

template<class T = void> struct equal_to {
    constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;

1

Returns: x == y.

template<> struct equal_to<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) == std::forward<U>(u));
    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) == std::forward<U>(u));

2

Returns: std::forward<T>(t) == std::forward<U>(u).

20.14.8.3 Class template not_equal_to

template<class T = void> struct not_equal_to {
    constexpr bool operator()(const T& x, const T& y) const;
};
constexpr bool operator()(const T& x, const T& y) const;

Returns: x != y.

template<> struct not_equal_to<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) != std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) != std::forward<U>(u));

2 Returns: std::forward<T>(t) != std::forward<U>(u).

20.14.8.4 Class template greater

 template<class T = void> struct greater {
    constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;

Returns: x > y.

template<> struct greater<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) > std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) > std::forward<U>(u));

2 Returns: std::forward<T>(t) > std::forward<U>(u).

20.14.8.5 Class template less

 template<class T = void> struct less {
    constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;

Returns: x < y.

template<> struct less<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) < std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) < std::forward<U>(u));

2 Returns: std::forward<T>(t) < std::forward<U>(u).

20.14.8.6 Class template greater_equal

 template<class T = void> struct greater_equal {
    constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;

Returns: x >= y.
template<class T, class U> constexpr bool operator()(const T& x, const T& y) const;

20.14.8.8 Class compare_three_way

In this subclause, BUILTIN-PTR-THREE-WAY(T, U) for types T and U is a boolean constant expression. BUILTIN-PTR-THREE-WAY(T, U) is true if and only if <=> in the expression
dclval<T>() <=> dclval<U>()
resolves to a built-in operator comparing pointers.

struct compare_three_way {
    template<class T, class U>
    requires three_way_comparable_with<T, U> || BUILTIN-PTR-THREE-WAY(T, U)
    constexpr auto operator()(T&& t, U&& u) const;

    using is_transparent = unspecified;
};

template<class T, class U>
requires three_way_comparable_with<T, U> || BUILTIN-PTR-THREE-WAY(T, U)
constexpr auto operator()(T&& t, U&& u) const;

2  Preconditions: If the expression std::forward<T>(t) <=> std::forward<U>(u) results in a call to a built-in operator <=> comparing pointers of type P, the conversion sequences from both T and U to P are equality-preserving (18.2).

3  Effects:

(3.1)  If the expression std::forward<T>(t) <=> std::forward<U>(u) results in a call to a built-in operator <=> comparing pointers of type P, returns strong_ordering::less if (the converted value of) t precedes u in the implementation-defined strict total order over pointers (3.27), strong_ordering::greater if u precedes t, and otherwise strong_ordering::equal.

(3.2)  Otherwise, equivalent to: return std::forward<T>(t) <=> std::forward<U>(u);
20.14.9  Concept-constrained comparisons

In this subclause, `BUILTIN-PTR-CMP(T, op, U)` for types T and U and where op is an equality (7.6.10) or relational operator (7.6.9) is a boolean constant expression. `BUILTIN-PTR-CMP(T, op, U)` is true if and only if op in the expression `declval<T>() op declval<U>()` resolves to a built-in operator comparing pointers.

```cpp
struct ranges::equal_to {
    template<class T, class U>
    requires equality_comparable_with<T, U> || BUILTIN-PTR-CMP(T, ==, U)
    constexpr bool operator()(T&& t, U&& u) const;

    using is_transparent = unspecified;
};
```

Preconditions: If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator == comparing pointers of type P, the conversion sequences from both T and U to P are equality-preserving (18.2).

Effects:

(3.1) — If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator == comparing pointers: returns false if either (the converted value of) t precedes u or u precedes t in the implementation-defined strict total order over pointers (3.27) and otherwise true.

(3.2) — Otherwise, equivalent to: return `std::forward<T>(t) == std::forward<U>(u)`;

```cpp
struct ranges::greater {
    template<class T, class U>
    requires totally_ordered_with<T, U> || BUILTIN-PTR-CMP(U, <, T)
    constexpr bool operator()(T&& t, U&& u) const;

    using is_transparent = unspecified;
};
```

operator() has effects equivalent to:

return !ranges::equal_to{}(std::forward<T>(t), std::forward<U>(u));

```cpp
struct ranges::less {
    template<class T, class U>
    requires totally_ordered_with<T, U> || BUILTIN-PTR-CMP(T, <, U)
    constexpr bool operator()(T&& t, U&& u) const;

    using is_transparent = unspecified;
};
```

operator() has effects equivalent to:

return ranges::less{}(std::forward<U>(u), std::forward<T>(t));

Effects:

(7.1) — If the expression `std::forward<T>(t) < std::forward<U>(u)` results in a call to a built-in operator < comparing pointers: returns true if (the converted value of) t precedes u in the implementation-defined strict total order over pointers (3.27) and otherwise false.
— Otherwise, equivalent to: \( \text{return std::forward<T>(t) < std::forward<U>(u);} \)

```cpp
struct ranges::greater_equal {
  template<class T, class U>
  requires totally_ordered_with<T, U> || BUILTIN-PTR-CMP(T, <, U)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

8 \( \text{operator()} \) has effects equivalent to:

\[
\text{return !ranges::less{}(std::forward<T>(t), std::forward<U>(u));}
\]

```cpp
struct ranges::less_equal {
  template<class T, class U>
  requires totally_ordered_with<T, U> || BUILTIN-PTR-CMP(U, <, T)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

9 \( \text{operator()} \) has effects equivalent to:

\[
\text{return !ranges::less{}(std::forward<U>(u), std::forward<T>(t));}
\]

### 20.14.10 Logical operations

#### 20.14.10.1 General

The library provides basic function object classes for all of the logical operators in the language (7.6.14, 7.6.15, 7.6.2.2).

#### 20.14.10.2 Class template logical_and

```cpp
template<class T = void> struct logical_and {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

1 \( \text{Returns: x && y} \).

```cpp
template<> struct logical_and<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) && std::forward<U>(u));

  using is_transparent = unspecified;
};
```

2 \( \text{Returns: std::forward<T>(t) && std::forward<U>(u).} \)

#### 20.14.10.3 Class template logical_or

```cpp
template<class T = void> struct logical_or {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

1 \( \text{Returns: x || y.} \)

```cpp
template<> struct logical_or<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) || std::forward<U>(u));
```
20.14.10.4 Class template logical_not

```cpp
template<class T, class U> constexpr auto operator()(T&& t, U&& u) const -> decltype(std::forward<T>(t) || std::forward<U>(u));
```

2 Returns: std::forward<T>(t) || std::forward<U>(u).

20.14.11 Bitwise operations

20.14.11.2 Class template bit_and

```cpp
template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) & std::forward<U>(u));
```

2 Returns: std::forward<T>(t) & std::forward<U>(u).

20.14.11.3 Class template bit_or

```cpp
template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) | std::forward<U>(u));
```

2 Returns: std::forward<T>(t) | std::forward<U>(u).
template<> struct bit_or<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) | std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
-> decltype(std::forward<T>(t) | std::forward<U>(u));

2 Returns: std::forward<T>(t) | std::forward<U>(u).

20.14.11.4 Class template bit_xor

2 Returns: std::forward<T>(t) | std::forward<U>(u).

template<struct T = void> struct bit_xor {
    constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;

1 Returns: x ^ y.

template<> struct bit_xor<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) ^ std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
-> decltype(std::forward<T>(t) ^ std::forward<U>(u));

2 Returns: std::forward<T>(t) ^ std::forward<U>(u).

20.14.11.5 Class template bit_not

2 Returns: ~std::forward<T>(t).

template<struct T = void> struct bit_not {
    constexpr T operator()(const T& x) const;
};

constexpr T operator()(const T& x) const;

1 Returns: ~x.

template<> struct bit_not<void> {
    template<class T> constexpr auto operator()(T&& t) const
    -> decltype(~std::forward<T>(t));

    using is_transparent = unspecified;
};

template<class T> constexpr auto operator()(T&&) const
-> decltype(~std::forward<T>(t));

2 Returns: ~std::forward<T>(t).

20.14.12 Class identity

1 Effects: Equivalent to: return std::forward<T>(t);

struct identity {
    template<class T>
    constexpr T&& operator()(T&& t) const noexcept;

    using is_transparent = unspecified;
};

template<class T>
constexpr T&& operator()(T&& t) const noexcept;

1
20.14.13 Function template not_fn

```cpp
template<class F> constexpr unspecified not_fn(F&& f);
```

1. In the text that follows:

- g is a value of the result of a not_fn invocation,
- FD is the type decay_t<F>,
- fd is the target object of g (20.14.3) of type FD, direct-non-list-initialized with std::forward<F>(f),
- call_args is an argument pack used in a function call expression (7.6.1.3) of g.

2. **Mandates**: `is_constructible_v<FD, F> && is_move_constructible_v<FD>` is true.
3. **Preconditions**: FD meets the Cpp17MoveConstructible requirements.
4. **Returns**: A perfect forwarding call wrapper g with call pattern `!invoke(fd, call_args...)`.
5. **Throws**: Any exception thrown by the initialization of fd.

20.14.14 Function template bind_front

```cpp
template<class F, class... Args>
constexpr unspecified bind_front(F&& f, Args&&... args);
```

1. In the text that follows:

- g is a value of the result of a bind_front invocation,
- FD is the type decay_t<F>,
- fd is the target object of g (20.14.3) of type FD, direct-non-list-initialized with std::forward<F>(f),
- BoundArgs is a pack that denotes decay_t<Args>...
- bound_args is a pack of bound argument entities of g (20.14.3) of types BoundArgs..., direction-non-list-initialized with std::forward<Args>(args)..., respectively, and
- call_args is an argument pack used in a function call expression (7.6.1.3) of g.

2. **Mandates**:

```
is_constructible_v<FD, F> &&
(is_constructible_v<BoundArgs, Args> && ...) &&
(is_move_constructible_v<BoundArgs> && ...)
```

is true.

3. **Preconditions**: FD meets the Cpp17MoveConstructible requirements. For each T_i in BoundArgs, if T_i is an object type, T_i meets the Cpp17MoveConstructible requirements.
4. **Returns**: A perfect forwarding call wrapper g with call pattern `invoke(fd, bound_args..., call_args...)`.
5. **Throws**: Any exception thrown by the initialization of the state entities of g (20.14.3).

20.14.15 Function object binders

20.14.15.1 General

Subclause 20.14.15 describes a uniform mechanism for binding arguments of callable objects.

20.14.15.2 Class template is_bind_expression

```cpp
namespace std {
    template<class T> struct is_bind_expression;  // see below
}
```

1. The class template is_bind_expression can be used to detect function objects generated by bind. The function template bind uses is_bind_expression to detect subexpressions.

2. Specializations of the is_bind_expression template shall meet the Cpp17UnaryTypeTrait requirements (20.15.2). The implementation provides a definition that has a base characteristic of true_type if T is a type
returned from bind, otherwise it has a base characteristic of \texttt{false\_type}. A program may specialize this template for a program-defined type \texttt{T} to have a base characteristic of \texttt{true\_type} to indicate that \texttt{T} should be treated as a subexpression in a \texttt{bind} call.

\textbf{20.14.15.3 Class template \texttt{is\_placeholder}}

\begin{verbatim}
namespace std {
    template<class T> struct is_placeholder;       // see below
}
\end{verbatim}

1. The class template \texttt{is\_placeholder} can be used to detect the standard placeholders \_\_1, \_\_2, and so on. The function template \texttt{bind} uses \texttt{is\_placeholder} to detect placeholders.

2. Specializations of the \texttt{is\_placeholder} template shall meet the \texttt{Cpp17UnaryTypeTrait} requirements (20.14.2). The implementation provides a definition that has the base characteristic of \texttt{integral\_constant\langle int, J\rangle} if \texttt{T} is the type of \texttt{std::placeholders::\_\_J}, otherwise it has a base characteristic of \texttt{integral\_constant\langle int, 0\rangle}. A program may specialize this template for a program-defined type \texttt{T} to have a base characteristic of \texttt{integral\_constant\langle int, N\rangle} with \texttt{N} \texttt{>} 0 to indicate that \texttt{T} should be treated as a placeholder type.

\textbf{20.14.15.4 Function template \texttt{bind}}

\begin{verbatim}
template<class F, class... BoundArgs>
constexpr unspecified bind(F&& f, BoundArgs&&... bound_args);
template<class R, class F, class... BoundArgs>
constexpr unspecified bind(F&& f, BoundArgs&&... bound_args);
\end{verbatim}

1. In the text that follows:

   (1.1) \textit{— g} is a value of the result of a \texttt{bind} invocation,

   (1.2) \textit{— FD} is the type \texttt{decay\_t\langle F\rangle},

   (1.3) \textit{— fd} is an lvalue that is a target object of \texttt{g} (20.14.3) of type \texttt{FD} direct-non-list-initialized with \texttt{std::forward\langle F\rangle\langle f\rangle},

   (1.4) \textit{— Ti} is the \texttt{i^th} type in the template parameter pack \texttt{BoundArgs},

   (1.5) \textit{— TDi} is the type \texttt{decay\_t\langle Ti\rangle},

   (1.6) \textit{— ti} is the \texttt{i^th} argument in the function parameter pack \texttt{bound\_args},

   (1.7) \textit{— td} is a bound argument entity of \texttt{g} (20.14.3) of type \texttt{TDi} direct-non-list-initialized with \texttt{std::forward\langle Ti\rangle\langle ti\rangle},

   (1.8) \textit{— Un} is the \texttt{j^th} deduced type of the \texttt{UnBoundArgs\&\&}... parameter of the argument forwarding call wrapper, and

   (1.9) \textit{— uN} is the \texttt{j^th} argument associated with \texttt{Un}.

2. \textbf{Mandates:} \texttt{is\_constructible\_v\langle FD, F\rangle} is \texttt{true}. For each \texttt{Ti} in \texttt{BoundArgs}, \texttt{is\_constructible\_v\langle TDi, Ti\rangle} is \texttt{true}.

3. \textbf{Preconditions:} \texttt{FD} and each \texttt{TDi} meet the \texttt{Cpp17Move\_Constructible} and \texttt{Cpp17Destructible} requirements. \texttt{INVOKE(fd, w1, w2, ..., wN)} (20.14.4) is a valid expression for some values \texttt{w1, w2, ..., wN}, where \texttt{N} has the value \texttt{sizeof...(bound\_args)}.

4. \textbf{Returns:} An argument forwarding call wrapper \texttt{g} (20.14.4). A program that attempts to invoke a volatile-qualified \texttt{g} is ill-formed. When \texttt{g} is not volatile-qualified, invocation of \texttt{g(u1, u2, ..., uM)} is expression-equivalent (3.21) to

\begin{verbatim}
INVOLVE \texttt{(static\_cast\langle V1\rangle\langle v1\rangle),
          static\_cast\langle V1\rangle\langle v1\rangle, static\_cast\langle V2\rangle\langle v2\rangle, ..., static\_cast\langle VN\rangle\langle vN\rangle)}
\end{verbatim}

for the first overload, and

\begin{verbatim}
INVOLVE\texttt{R}\texttt{(static\_cast\langle V1\rangle\langle v1\rangle),
          static\_cast\langle V1\rangle\langle v1\rangle, static\_cast\langle V2\rangle\langle v2\rangle, ..., static\_cast\langle VN\rangle\langle vN\rangle)}
\end{verbatim}

for the second overload, where the values and types of the target argument \texttt{v\_fd} and of the bound arguments \texttt{v1, v2, ..., vN} are determined as specified below.

5. \textbf{Throws:} Any exception thrown by the initialization of the state entities of \texttt{g}.

6. \textit{[Note 1:] If all of \texttt{FD} and \texttt{TD} meet the requirements of \texttt{Cpp17Copy\_Constructible}, then the return type meets the requirements of \texttt{Cpp17Copy\_Constructible}. — end note]
The values of the bound arguments \( v_1, v_2, \ldots, v_N \) and their corresponding types \( V_1, V_2, \ldots, V_N \) depend on the types \( TD_i \) derived from the call to \texttt{bind} and the cv-qualifiers \( cv \) of the call wrapper \( g \) as follows:

1. If \( TD_i \) is \texttt{reference_wrapper<T>} and the \texttt{cv} is \texttt{true}, the argument is \( td_i\.get() \) and its type \( V_i \) is \( T\.\& \).
2. If the value of \texttt{is\_bind\_expression\_v<TD>\_i>} is \texttt{true}, the argument is
   \[
   \text{static\_cast<cv TD_i\&>(td_i\(_\ldots\)u_j\(_\ldots\))}
   \]
   and its type \( V_i \) is \( \text{void\_cv\_t<cv TD_i\&, U_j\(_\ldots\)>\&\&} \).
3. If the value \( j \) of \texttt{is\_placeholder\_v<TD>\_i>} is not zero, the argument is \( std::forward\<U_j\(_\ldots\)>\(_\ldots\)) \) and its type \( V_i \) is \( U_j\&\& \).
4. Otherwise, the value is \( td_i \) and its type \( V_i \) is \( cv TD_i\& \).

The value of the target argument \( v_{fd} \) is \( fd \) and its corresponding type \( V_{fd} \) is \( cv FD\& \).

20.14.15.5 Placeholders

namespace std::placeholders {
    // M is the implementation-defined number of placeholders
    see below _1;
    see below _2;
    .
    .
    .
    see below _M;
}

All placeholder types meet the \texttt{Cpp17DefaultConstructible} and \texttt{Cpp17CopyConstructible} requirements, and their default constructors and copy/move constructors are constexpr functions that do not throw exceptions. It is implementation-defined whether placeholder types meet the \texttt{Cpp17CopyAssignable} requirements, but if so, their copy assignment operators are constexpr functions that do not throw exceptions.

Placeholders should be defined as:

\begin{verbatim}
inline constexpr unspecified _1{};
\end{verbatim}

If they are not, they are declared as:

\begin{verbatim}
extern unspecified _1;
\end{verbatim}

20.14.16 Function template \texttt{mem\_fn}

\begin{verbatim}
template<class R, class T> constexpr unspecified mem\_fn(R T\::* pm) noexcept;
\end{verbatim}

Returns: A simple call wrapper (20.14.3) \( fn \) with call pattern \texttt{invoke(pm, \ldots)} where \( pm \) is the target object of \( fn \) of type \( R T\::* \) direct-non-list-initialized with \( pm \), and \( \ldots \) is an argument pack used in a function call expression (7.6.1.3) of \( pm \).

20.14.17 Polymorphic function wrappers

20.14.17.1 General

Subclause 20.14.17 describes a polymorphic wrapper class that encapsulates arbitrary callable objects.

20.14.17.2 Class \texttt{bad\_function\_call}

An exception of type \texttt{bad\_function\_call} is thrown by \texttt{function::operator()} (20.14.17.3.5) when the function wrapper object has no target.

\begin{verbatim}
namespace std {
    class bad\_function\_call : public exception {
        public:
            // see 17.9.3 for the specification of the special member functions
            const char* what() const noexcept override;
    };
}
\end{verbatim}

Returns: An implementation-defined \texttt{ntbs}.
20.14.17.3  Class template function

20.14.17.3.1  General

namespace std {

    template<class> class function;  // not defined

    template<class R, class... ArgTypes>
    class function<R(ArgTypes...)>
    {
        public:
            using result_type = R;

            // 20.14.17.3.2, construct/copy/destroy
            function() noexcept;
            function(nullptr_t) noexcept;
            function(const function&);
            function(function&&) noexcept;
            template<class F>
            function(F);
            function& operator=(const function&);
            function& operator=(function&&);
            template<class F>
            function& operator=(F&&);
            template<class F>
            function& operator=(reference_wrapper<F>) noexcept;
            ~function();

            // 20.14.17.3.3, function modifiers
            void swap(function&) noexcept;

            // 20.14.17.3.4, function capacity
            explicit operator bool() const noexcept;

            // 20.14.17.3.5, function invocation
            R operator()(ArgTypes...) const;

            // 20.14.17.3.6, function target access
            const type_info& target_type() const noexcept;
            template<class T>
            T* target() noexcept;
            template<class T>
            const T* target() const noexcept;
    };

    template<class R, class... ArgTypes>
    function(R(*)(ArgTypes...)) -> function<R(ArgTypes...)>

    template<class F>
    function(F) -> function<see below>;

    // 20.14.17.3.7, null pointer comparison operator functions
    template<class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

    // 20.14.17.3.8, specialized algorithms
    template<class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;

}  // namespace std

1. The function class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects (20.14.3), given a call signature (20.14.3), allowing functions to be first-class objects.

2. A callable type (20.14.3) F is Lvalue-Callable for argument types ArgTypes and return type R if the expression INVOKE<R>(declval<F&>(), declval<ArgTypes>()), considered as an unevaluated operand (7.2), is well-formed (20.14.4).

3. The function class template is a call wrapper (20.14.3) whose call signature (20.14.3) is R(ArgTypes...).
20.14.17.3.2 Constructors and destructor

function() noexcept;

Postconditions: !*this.

function(nullptr_t) noexcept;

Postconditions: !*this.

function(const function& f);

Postconditions: !*this if !f; otherwise, *this targets a copy of f.target().

Throws: Nothing if f's target is a specialization of reference_wrapper or a function pointer. Otherwise, may throw bad_alloc or any exception thrown by the copy constructor of the stored callable object.

Recommended practice: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where f’s target is an object holding only a pointer or reference to an object and a member function pointer.

function(function&& f) noexcept;

Postconditions: If !f, *this has no target; otherwise, the target of *this is equivalent to the target of f before the construction, and f is in a valid state with an unspecified value.

Recommended practice: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where f’s target is an object holding only a pointer or reference to an object and a member function pointer.

template<class F> function(F f);

Constraints: F is Lvalue-Callable (20.14.17.3) for argument types ArgTypes... and return type R.

Preconditions: F meets the Cpp17CopyConstructible requirements.

Postconditions: !*this if any of the following hold:

1. f is a null function pointer value.
2. f is a null member pointer value.
3. F is an instance of the function class template, and !f.

Otherwise, *this targets a copy of f initialized with std::move(f).

Throws: Nothing if f is a specialization of reference_wrapper or a function pointer. Otherwise, may throw bad_alloc or any exception thrown by F’s copy or move constructor.

Recommended practice: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where f is an object holding only a pointer or reference to an object and a member function pointer.

template<class F> function(F) -> function<see below>;

Constraints: &F::operator() is well-formed when treated as an unevaluated operand and decltype(&F::operator()) is of the form R(G::*)(A...) cv kopt noexcept opt for a class type G.

Remarks: The deduced type is function<R(A...)>

[Example 1:]

```cpp
void f() {
  int i(5);
  function g = [&](double) { return i; };  // deduces function<int(double)>
}
```

Returns: *this.

function& operator=(const function& f);

Effects: As if by function(f).swap(*this);

Returns: *this.
function& operator=(function&& f);
          \textit{Effects}: Replaces the target of \texttt{*this} with the target of \texttt{f}.
          \textit{Returns}: \texttt{*this}.

function& operator=(nullptr_t) noexcept;
          \textit{Effects}: If \texttt{*this} \neq \texttt{nullptr}, destroys the target of \texttt{this}.
          \textit{Postconditions}: !(\texttt{*this}).
          \textit{Returns}: \texttt{*this}.

template<class F> function& operator=(F&& f);
          \textit{Constraints}: decay_t<F> is Lvalue-Callable (20.14.17.3) for argument types \texttt{ArgTypes}... and return type \texttt{R}.
          \textit{Effects}: As if by: function(std::forward<F>(f)).swap(*this);
          \textit{Returns}: \texttt{*this}.

template<class F> function& operator=(reference_wrapper<F> f) noexcept;
          \textit{Effects}: As if by: function(f).swap(*this);
          \textit{Returns}: \texttt{*this}.

~function();
          \textit{Effects}: If \texttt{*this} \neq \texttt{nullptr}, destroys the target of \texttt{this}.

20.14.17.3.3 Modifiers

void swap(function& other) noexcept;
          \textit{Effects}: Interchanges the targets of \texttt{*this} and \texttt{other}.

20.14.17.3.4 Capacity

explicit operator bool() const noexcept;
          \textit{Returns}: true if \texttt{*this} has a target, otherwise false.

20.14.17.3.5 Invocation

R operator()(ArgTypes... args) const;
          \textit{Returns}: INVOKE<R>(f, std::forward<ArgTypes>(args)...) (20.14.4), where \texttt{f} is the target object (20.14.3) of \texttt{*this}.
          \textit{Throws}: bad_function_call if \texttt{!*this}; otherwise, any exception thrown by the wrapped callable object.

20.14.17.3.6 Target access

const type_info& target_type() const noexcept;
          \textit{Returns}: If \texttt{*this} has a target of type \texttt{T}, \texttt{typeid(T)}; otherwise, \texttt{typeid(void)}.

template<class T> T* target() noexcept;

template<class T> const T* target() const noexcept;
          \textit{Returns}: If \texttt{target_type()} \texttt{== typeid(T)} a pointer to the stored function target; otherwise a null pointer.

20.14.17.3.7 Null pointer comparison operator functions

template<class R, class... ArgTypes>
bool operator==(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;
          \textit{Returns}: \texttt{!f}.
20.14.17.3.8 Specialized algorithms

```cpp
template<class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2) noexcept;
```

1. *Effects*: As if by: `f1.swap(f2);`

20.14.18 Searchers

20.14.18.1 General

1. Subclause 20.14.18 provides function object types (20.14) for operations that search for a sequence `[pat_first, pat_last)` in another sequence `[first, last)` that is provided to the object’s function call operator. The first sequence (the pattern to be searched for) is provided to the object’s constructor, and the second (the sequence to be searched) is provided to the function call operator.

2. Each specialization of a class template specified in 20.14.18 shall meet the *Cpp17CopyConstructible* and *Cpp17CopyAssignable* requirements. Template parameters named

   - *(2.1)* `ForwardIterator`,
   - *(2.2)* `ForwardIterator1`,
   - *(2.3)* `ForwardIterator2`,
   - *(2.4)* `RandomAccessIterator`,
   - *(2.5)* `RandomAccessIterator1`,
   - *(2.6)* `RandomAccessIterator2`, and
   - *(2.7)* `BinaryPredicate`

   of templates specified in 20.14.18 shall meet the same requirements and semantics as specified in 25.1. Template parameters named `Hash` shall meet the *Cpp17Hash* requirements (Table 34).

3. The Boyer-Moore searcher implements the Boyer-Moore search algorithm. The Boyer-Moore-Horspool searcher implements the Boyer-Moore-Horspool search algorithm. In general, the Boyer-Moore searcher will use more memory and give better runtime performance than Boyer-Moore-Horspool.

20.14.18.2 Class template `default_searcher`

```cpp
template<class ForwardIterator1, class BinaryPredicate = equal_to<>>
class default_searcher {
public:
  constexpr default_searcher(ForwardIterator1 pat_first, ForwardIterator1 pat_last,
                             BinaryPredicate pred = BinaryPredicate());

  template<class ForwardIterator2>
  constexpr pair<ForwardIterator2, ForwardIterator2>
  operator()(ForwardIterator2 first, ForwardIterator2 last) const;

private:
  ForwardIterator1 pat_first_;       // exposition only
  ForwardIterator1 pat_last_;        // exposition only
  BinaryPredicate pred_;             // exposition only
};
```

1. *Effects*: Constructs a `default_searcher` object, initializing `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, and `pred_` with `pred`.

2. *Throws*: Any exception thrown by the copy constructor of `BinaryPredicate` or `ForwardIterator1`.

```cpp
template<class ForwardIterator2>
constexpr pair<ForwardIterator2, ForwardIterator2>
operator()(ForwardIterator2 first, ForwardIterator2 last) const;
```

1. *Effects*: Returns a pair of iterators `i` and `j` such that

   - `i == search(first, last, pat_first_, pat_last_, pred_, and`
— if \( i == \text{last} \), then \( j == \text{last} \), otherwise \( j == \text{next}(i, \text{distance}(\text{pat_first}, \text{pat_last})) \).

### 20.14.18.4 Class template `boyer_moore_searcher`

```cpp
template<class RandomAccessIterator1,
         class Hash = hash<typename iterator_traits<RandomAccessIterator1>::value_type>,
         class BinaryPredicate = equal_to<>>
class boyer_moore_searcher {
  public:
    boyer_moore_searcher(RandomAccessIterator1 pat_first,
                         RandomAccessIterator1 pat_last,
                         Hash hf = Hash(),
                         BinaryPredicate pred = BinaryPredicate());
  
    template<class RandomAccessIterator2>
    pair<RandomAccessIterator2, RandomAccessIterator2>
    operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;

  private:
    RandomAccessIterator1 pat_first_; // exposition only
    RandomAccessIterator1 pat_last_;  // exposition only
    Hash hash_;                      // exposition only
    BinaryPredicate pred_;           // exposition only

    boyer_moore_searcher(RandomAccessIterator1 pat_first,
                          RandomAccessIterator1 pat_last,
                          Hash hf = Hash(),
                          BinaryPredicate pred = BinaryPredicate());

Preconditions: The value type of `RandomAccessIterator1` meets the `Cpp17DefaultConstructible` requirements, the `Cpp17CopyConstructible` requirements, and the `Cpp17CopyAssignable` requirements.

Let \( V \) be `iterator_traits<RandomAccessIterator1>::value_type`. For any two values \( A \) and \( B \) of type \( V \), if \( \text{pred}(A, B) == \text{true} \), then \( \text{hf}(A) == \text{hf}(B) \) is true.

Effects: Initializes `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, `hash_` with `hf`, and `pred_` with `pred`.

Throws: Any exception thrown by the copy constructor of `RandomAccessIterator1`, or by the default constructor, copy constructor, or the copy assignment operator of the value type of `RandomAccessIterator1`, or the copy constructor or `operator()` of `BinaryPredicate` or `Hash`. May throw `bad_alloc` if additional memory needed for internal data structures cannot be allocated.

```
public:
    boyer_moore_horspool_searcher(RandomAccessIterator1 pat_first,
        RandomAccessIterator1 pat_last,
        Hash hf = Hash(),
        BinaryPredicate pred = BinaryPredicate());

    template<class RandomAccessIterator2>
    pair<RandomAccessIterator2, RandomAccessIterator2>
        operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;

private:
    RandomAccessIterator1 pat_first_; // exposition only
    RandomAccessIterator1 pat_last_;   // exposition only
    Hash hash_;                       // exposition only
    BinaryPredicate pred_;            // exposition only

    boyer_moore_horspool_searcher(RandomAccessIterator1 pat_first,
        RandomAccessIterator1 pat_last,
        Hash hf = Hash(),
        BinaryPredicate pred = BinaryPredicate());

Preconditions: The value type of RandomAccessIterator1 meets the Cpp17DefaultConstructible,
                Cpp17CopyConstructible, and Cpp17CopyAssignable requirements.

Let V be iterator_traits<RandomAccessIterator1>::value_type. For any two values A and B of
    type V, if pred(A, B) == true, then hf(A) == hf(B) is true.

Effects: Initializes pat_first_ with pat_first, pat_last_ with pat_last, hash_ with hf, and pred_
          with pred.

Throws: Any exception thrown by the copy constructor of RandomAccessIterator1, or by the default
         constructor, copy constructor, or the copy assignment operator of the value type of RandomAccess-
         Iterator1 or the copy constructor or operator() of BinaryPredicate or Hash. May throw bad_alloc
         if additional memory needed for internal data structures cannot be allocated.

    template<class RandomAccessIterator2>
    pair<RandomAccessIterator2, RandomAccessIterator2>
        operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;

    Mandates: RandomAccessIterator1 and RandomAccessIterator2 have the same value type.

    Effects: Finds a subsequence of equal values in a sequence.

    Returns: A pair of iterators i and j such that

(7.1) — i is the first iterator i in the range [first, last - (pat_last_ - pat_first_)) such that for
         every non-negative integer n less than pat_last_ - pat_first_ the following condition holds:
         pred(*(i + n), *(pat_first_ + n)) != false, and

(7.2) — j == next(i, distance(pat_first_, pat_last_)).

    Returns make_pair(first, first) if [pat_first_, pat_last_) is empty, otherwise returns make_-
    pair(last, last) if no such iterator is found.

    Complexity: At most (last - first) * (pat_last_ - pat_first_) applications of the predicate.

20.14.19  Class template hash

The unordered associative containers defined in 22.5 use specializations of the class template hash (20.14.2) as
the default hash function.

Each specialization of hash is either enabled or disabled, as described below.

[Note 1: Enabled specializations meet the Cpp17Hash requirements, and disabled specializations do not. — end note]

Each header that declares the template hash provides enabled specializations of hash for nullptr_t and all
cv-unqualified arithmetic, enumeration, and pointer types. For any type Key for which neither the library
nor the user provides an explicit or partial specialization of the class template hash, hash<Key> is disabled.

If the library provides an explicit or partial specialization of hash<Key>, that specialization is enabled except
as noted otherwise, and its member functions are noexcept except as noted otherwise.
If $H$ is a disabled specialization of `hash`, these values are false:\[ is_default_constructible_v<H>, is_copy_constructible_v<H>, is_move_constructible_v<H>, is_copy_assignable_v<H>, and is_move_assignable_v<H>\]. Disabled specializations of `hash` are not function object types (20.14).

[Note 2: This means that the specialization of `hash` exists, but any attempts to use it as a `Cpp17Hash` will be ill-formed. — end note]

An enabled specialization `hash<Key>` will:

(5.1)  meet the `Cpp17Hash` requirements (Table 34), with `Key` as the function call argument type, the `Cpp17DefaultConstructible` requirements (Table 27), the `Cpp17CopyAssignable` requirements (Table 31),

(5.2)  be swappable (16.4.4.3) for lvalues,

(5.3)  meet the requirement that if $k1 == k2$ is true, $h(k1) == h(k2)$ is also true, where $h$ is an object of type `hash<Key>` and $k1$ and $k2$ are objects of type `Key`;

(5.4)  meet the requirement that the expression $h(k)$, where $h$ is an object of type `hash<Key>` and $k$ is an object of type `Key`, shall not throw an exception unless `hash<Key>` is a program-defined specialization that depends on at least one program-defined type.

## 20.15 Metaprogramming and type traits

### 20.15.1 General

Subclause 20.15 describes components used by C++ programs, particularly in templates, to support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time. It includes type classification traits, type property inspection traits, and type transformations. The type classification traits describe a complete taxonomy of all possible C++ types, and state where in that taxonomy a given type belongs. The type property inspection traits allow important characteristics of types or of combinations of types to be inspected. The type transformations allow certain properties of types to be manipulated.

All functions specified in 20.15 are signal-safe (17.13.5).

### 20.15.2 Requirements

A `Cpp17UnaryTypeTrait` describes a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the property being described. It shall be `Cpp17DefaultConstructible`, `Cpp17CopyConstructible`, and publicly and unambiguously derived, directly or indirectly, from its base characteristic, which is a specialization of the template `integral_constant` (20.15.4), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described. The member names of the base characteristic shall not be hidden and shall be unambiguously available in the `Cpp17UnaryTypeTrait`.

A `Cpp17BinaryTypeTrait` describes a relationship between two types. It shall be a class template that takes two template type arguments and, optionally, additional arguments that help define the relationship being described. It shall be `Cpp17DefaultConstructible`, `Cpp17CopyConstructible`, and publicly and unambiguously derived, directly or indirectly, from its base characteristic, which is a specialization of the template `integral_constant` (20.15.4), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described. The member names of the base characteristic shall not be hidden and shall be unambiguously available in the `Cpp17BinaryTypeTrait`.

A `Cpp17TransformationTrait` modifies a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the modification. It shall define a publicly accessible nested type named `type`, which shall be a synonym for the modified type.

Unless otherwise specified, the behavior of a program that adds specializations for any of the templates specified in 20.15 is undefined.

Unless otherwise specified, an incomplete type may be used to instantiate a template specified in 20.15. The behavior of a program is undefined if:

(5.1)  an instantiation of a template specified in 20.15 directly or indirectly depends on an incompletely-defined object type $T$, and

(5.2)  that instantiation could yield a different result were $T$ hypothetically completed.
20.15.3 Header `<type_traits>` synopsis

```cpp
namespace std {
  // 20.15.4, helper class
template<class T, T v> struct integral_constant;

  template<bool B>
  using bool_constant = integral_constant<bool, B>;
  using true_type = bool_constant<true>;
  using false_type = bool_constant<false>;

  // 20.15.5.2, primary type categories
  template<class T> struct is_void;
  template<class T> struct is_null_pointer;
  template<class T> struct is_integral;
  template<class T> struct is_floating_point;
  template<class T> struct is_array;
  template<class T> struct is_pointer;
  template<class T> struct is_lvalue_reference;
  template<class T> struct is_rvalue_reference;
  template<class T> struct is_member_object_pointer;
  template<class T> struct is_member_function_pointer;
  template<class T> struct is_enum;
  template<class T> struct is_union;
  template<class T> struct is_class;
  template<class T> struct is_function;

  // 20.15.5.3, composite type categories
  template<class T> struct is_reference;
  template<class T> struct is_arithmetic;
  template<class T> struct is_fundamental;
  template<class T> struct is_object;
  template<class T> struct is_scalar;
  template<class T> struct is_compound;
  template<class T> struct is_member_pointer;

  // 20.15.5.4, type properties
  template<class T> struct is_const;
  template<class T> struct is_volatile;
  template<class T> struct is_trivial;
  template<class T> struct is_trivially_copyable;
  template<class T> struct is_standard_layout;
  template<class T> struct is_empty;
  template<class T> struct is_polymorphic;
  template<class T> struct is_abstract;
  template<class T> struct is_final;
  template<class T> struct is_aggregate;

  template<class T> struct is_signed;
  template<class T> struct is_unsigned;
  template<class T> struct is_bounded_array;
  template<class T> struct is_unbounded_array;

  template<class T, class... Args> struct is_constructible;
  template<class T> struct is_default_constructible;
  template<class T> struct is_copy_constructible;
  template<class T> struct is_move_constructible;

  template<class T, class U> struct is_assignable;
  template<class T> struct is_copy_assignable;
  template<class T> struct is_move_assignable;

  template<class T, class U> struct is_swappable_with;
  template<class T> struct is_swappable;
```
template<class T> struct is_destructible;

template<class T, class... Args> struct is_trivially_constructible;

template<class T> struct is_trivially_default_constructible;

template<class T> struct is_trivially_copy_constructible;

template<class T> struct is_trivially_move_constructible;

template<class T, class U> struct is_trivially_assignable;

template<class T> struct is_trivially_copy_assignable;

template<class T> struct is_trivially_move_assignable;

template<class T> struct is_trivially_destructible;

template<class T, class... Args> struct is_nothrow_constructible;

template<class T> struct is_nothrow_default_constructible;

template<class T> struct is_nothrow_copy_constructible;

template<class T> struct is_nothrow_move_constructible;

template<class T, class U> struct is_nothrow_assignable;

template<class T> struct is_nothrow_copy_assignable;

template<class T> struct is_nothrow_move_assignable;

template<class T, class U> struct is_nothrow_swappable_with;

template<class T> struct is_nothrow_swappable;

template<class T> struct is_nothrow_destructible;

template<class T> struct has_virtual_destructor;

template<class T> struct has_unique_object_representations;

// 20.15.6, type property queries
template<class T> struct alignment_of;

template<class T> struct rank;

template<class T, unsigned I = 0> struct extent;

// 20.15.7, type relations
template<class T, class U> struct is_same;

template<class Base, class Derived> struct is_base_of;

template<class From, class To> struct is_convertible;

template<class From, class To> struct is_nothrow_convertible;

template<class T, class U> struct is_layout_compatible;

template<class Base, class Derived> struct is_pointer_interconvertible_base_of;

template<class Fn, class... ArgTypes> struct is_invocable;

template<class R, class Fn, class... ArgTypes> struct is_invocable_r;

template<class Fn, class... ArgTypes> struct is_nothrow_invocable;

template<class R, class Fn, class... ArgTypes> struct is_nothrow_invocable_r;

// 20.15.8.2, const-volatile modifications
template<class T> struct remove_const;

template<class T> struct remove_volatile;

template<class T> struct remove_cv;

template<class T> struct add_const;

template<class T> struct add_volatile;

template<class T> struct add_cv;

template<class T>
using remove_const_t = typename remove_const<T>::type;

template<class T>
using remove_volatile_t = typename remove_volatile<T>::type;

template<class T>
using remove_cv_t = typename remove_cv<T>::type;
template<class T>
using add_const_t = typename add_const<T>::type;

template<class T>
using add_volatile_t = typename add_volatile<T>::type;

template<class T>
using add_cv_t = typename add_cv<T>::type;

// 20.15.8.3, reference modifications
template<class T> struct remove_reference;

template<class T> struct add_lvalue_reference;

template<class T> struct add_rvalue_reference;

template<class T>
using remove_reference_t = typename remove_reference<T>::type;

template<class T>
using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;

template<class T>
using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;

// 20.15.8.4, sign modifications
template<class T> struct make_signed;

template<class T> struct make_unsigned;

template<class T>
using make_signed_t = typename make_signed<T>::type;

template<class T>
using make_unsigned_t = typename make_unsigned<T>::type;

// 20.15.8.5, array modifications
template<class T> struct remove_extent;

template<class T> struct remove_all_extents;

template<class T>
using remove_extent_t = typename remove_extent<T>::type;

template<class T>
using remove_all_extents_t = typename remove_all_extents<T>::type;

// 20.15.8.6, pointer modifications
template<class T> struct remove_pointer;

template<class T> struct add_pointer;

template<class T>
using remove_pointer_t = typename remove_pointer<T>::type;

template<class T>
using add_pointer_t = typename add_pointer<T>::type;

// 20.15.8.7, other transformations
template<class T> struct type_identity;

template<size_t Len, size_t Align = default_alignment> // see 20.15.8.7
struct aligned_storage;

template<size_t Len, class... Types> struct aligned_union;

template<class T> struct remove_cvref;

template<class T> struct decay;

template<bool, class T = void> struct enable_if;

template<bool, class T, class F> struct conditional;

template<class... T> struct common_type;

template<class T, class U, template<class> class TQual, template<class> class UQual>
struct basic_common_reference { };

template<class T, class U, template<class> class TQual, template<class> class UQual>
struct common_reference;

template<class T> struct underlying_type;

template<class Fn, class... ArgTypes> struct invoke_result;

template<class T> struct unwrap_reference;

template<class T> struct unwrap_ref_decay;
template<
class T>
using type_identity_t = typename type_identity<T>::type;
template<
size_t Len, size_t Align = default_alignment>
using aligned_storage_t = typename aligned_storage<Len, Align>::type;
template<
size_t Len, class... Types>
using aligned_union_t = typename aligned_union<Len, Types...>::type;
template<class T>
using remove_cvref_t = typename remove_cvref<T>::type;
template<class T>
using decay_t = typename decay<T>::type;
template<
bool b, class T = void>
using enable_if_t = typename enable_if<b, T>::type;
template<
bool b, class T, class F>
using conditional_t = typename conditional<b, T, F>::type;
template<class... T>
using common_type_t = typename common_type<T...>::type;
template<class... T>
using common_reference_t = typename common_reference<T...>::type;
template<class T>
using underlying_type_t = typename underlying_type<T>::type;
template<class Fn, class... ArgTypes>
using invoke_result_t = typename invoke_result<Fn, ArgTypes...>::type;
template<class T>
using unwrap_reference_t = typename unwrap_reference<T>::type;
template<class T>
using unwrap_ref_decay_t = typename unwrap_ref_decay<T>::type;
template<class... T>
using void_t = void;

§ 20.15.3

// 20.15.3, logical operator traits

template<class... B> struct conjunction;
template<class... B> struct disjunction;
template<class B> struct negation;

// 20.15.5.2, primary type categories

template<class T>
inline constexpr bool is_void_v = is_void<T>::value;
template<class T>
inline constexpr bool is_null_pointer_v = is_null_pointer<T>::value;
template<class T>
inline constexpr bool is_integral_v = is_integral<T>::value;
template<class T>
inline constexpr bool is_floating_point_v = is_floating_point<T>::value;
template<class T>
inline constexpr bool is_array_v = is_array<T>::value;
template<class T>
inline constexpr bool is_pointer_v = is_pointer<T>::value;
template<class T>
inline constexpr bool is_lvalue_reference_v = is_lvalue_reference<T>::value;
template<class T>
inline constexpr bool is_rvalue_reference_v = is_rvalue_reference<T>::value;
template<class T>
inline constexpr bool is_member_object_pointer_v = is_member_object_pointer<T>::value;
template<class T>
inline constexpr bool is_member_function_pointer_v = is_member_function_pointer<T>::value;
template<class T>
inline constexpr bool is_enum_v = is_enum<T>::value;
template<class T>
inline constexpr bool is_union_v = is_union<T>::value;
template<class T>
inline constexpr bool is_class_v = is_class<T>::value;
template<class T>
inline constexpr bool is_function_v = is_function<T>::value;
// 20.15.5.3, composite type categories
template<class T>
inline constexpr bool is_reference_v = is_reference<T>::value;
template<class T>
inline constexpr bool is_arithmetic_v = is_arithmetic<T>::value;
template<class T>
inline constexpr bool is_fundamental_v = is_fundamental<T>::value;
template<class T>
inline constexpr bool is_object_v = is_object<T>::value;
template<class T>
inline constexpr bool is_scalar_v = is_scalar<T>::value;
template<class T>
inline constexpr bool is_compound_v = is_compound<T>::value;
template<class T>
inline constexpr bool is_member_pointer_v = is_member_pointer<T>::value;

// 20.15.5.4, type properties
template<class T>
inline constexpr bool is_const_v = is_const<T>::value;
template<class T>
inline constexpr bool is_volatile_v = is_volatile<T>::value;
template<class T>
inline constexpr bool is_trivial_v = is_trivial<T>::value;
template<class T>
inline constexpr bool is_trivially_copyable_v = is_trivially_copyable<T>::value;
template<class T>
inline constexpr bool is_standard_layout_v = is_standard_layout<T>::value;
template<class T>
inline constexpr bool is_empty_v = is_empty<T>::value;
template<class T>
inline constexpr bool is_polymorphic_v = is_polymorphic<T>::value;
template<class T>
inline constexpr bool is_final_v = is_final<T>::value;
template<class T>
inline constexpr bool is_aggregate_v = is_aggregate<T>::value;
template<class T>
inline constexpr bool is_signed_v = is_signed<T>::value;
template<class T>
inline constexpr bool is_unsigned_v = is_unsigned<T>::value;
template<class T>
inline constexpr bool is_bounded_array_v = is_bounded_array<T>::value;
template<class T>
inline constexpr bool is_unbounded_array_v = is_unbounded_array<T>::value;
template<class T, class... Args>
inline constexpr bool is_constructible_v = is_constructible<T, Args...>::value;
template<class T>
inline constexpr bool is_default_constructible_v = is_default_constructible<T>::value;
template<class T>
inline constexpr bool is_copy_constructible_v = is_copy_constructible<T>::value;
template<class T>
inline constexpr bool is_move_constructible_v = is_move_constructible<T>::value;
template<class T, class U>
inline constexpr bool is_assignable_v = is_assignable<T, U>::value;
template<class T>
inline constexpr bool is_copy_assignable_v = is_copy_assignable<T>::value;
template<class T>
inline constexpr bool is_move_assignable_v = is_move_assignable<T>::value;
template<class T, class U>
inline constexpr bool is_swappable_with_v = is_swappable_with<T, U>::value;
template<class T>
inline constexpr bool is_swappable_v = is_swappable<T>::value;
template<class T>
inline constexpr bool is_destructible_v = is_destructible<T>::value;

template<class T, class... Args>
inline constexpr bool is_trivially_constructible_v
= is_trivially_constructible<T, Args...>::value;

template<class T>
inline constexpr bool is_trivially_default_constructible_v
= is_trivially_default_constructible<T>::value;

template<class T>
inline constexpr bool is_trivially_copy_constructible_v
= is_trivially_copy_constructible<T>::value;

template<class T>
inline constexpr bool is_trivially_move_constructible_v
= is_trivially_move_constructible<T>::value;

template<class T, class U>
inline constexpr bool is_trivially_assignable_v = is_trivially_assignable<T, U>::value;

template<class T>
inline constexpr bool is_trivially_copy_assignable_v
= is_trivially_copy_assignable<T>::value;

template<class T>
inline constexpr bool is_trivially_move_assignable_v
= is_trivially_move_assignable<T>::value;

template<class T>
inline constexpr bool is_trivially_destructible_v = is_trivially_destructible<T>::value;

template<class T, class... Args>
inline constexpr bool is_nothrow_constructible_v
= is_nothrow_constructible<T, Args...>::value;

template<class T>
inline constexpr bool is_nothrow_default_constructible_v
= is_nothrow_default_constructible<T>::value;

template<class T>
inline constexpr bool is_nothrow_copy_constructible_v
= is_nothrow_copy_constructible<T>::value;

template<class T>
inline constexpr bool is_nothrow_move_constructible_v
= is_nothrow_move_constructible<T>::value;

template<class T, class U>
inline constexpr bool is_nothrow_assignable_v = is_nothrow_assignable<T, U>::value;

template<class T>
inline constexpr bool is_nothrow_copy_assignable_v = is_nothrow_copy_assignable<T>::value;

template<class T>
inline constexpr bool is_nothrow_move_assignable_v = is_nothrow_move_assignable<T>::value;

template<class T, class U>
inline constexpr bool is_nothrow_swappable_with_v = is_nothrow_swappable_with<T, U>::value;

template<class T>
inline constexpr bool is_nothrow_swappable_v = is_nothrow_swappable<T>::value;

template<class T>
inline constexpr bool is_nothrow_destructible_v = is_nothrow_destructible<T>::value;

template<class T>
inline constexpr bool has_virtual_destructor_v = has_virtual_destructor<T>::value;

template<class T>
inline constexpr bool has_unique_object_representations_v
= has_unique_object_representations<T>::value;

// 20.15.6, type property queries

template<class T>
inline constexpr size_t alignment_of_v = alignment_of<T>::value;

template<class T>
inline constexpr size_t rank_v = rank<T>::value;

template<class T, unsigned I = 0>
inline constexpr size_t extent_v = extent<T, I>::value;
The class template `integral_constant`, alias template `bool_constant`, and its associated `typedef-names` `true_type` and `false_type` are used as base classes to define the interface for various type traits.

Subclause 20.15.5 contains templates that may be used to query the properties of a type at compile time.

Each of these templates shall be a C++17UnaryTypeTrait (20.15.2) with a base characteristic of `true_type` if the corresponding condition is `true`, otherwise `false_type`. 

1 The class template `integral_constant`, alias template `bool_constant`, and its associated `typedef-names` `true_type` and `false_type` are used as base classes to define the interface for various type traits.
20.15.5.2 Primary type categories

1 The primary type categories correspond to the descriptions given in subclause 6.8 of the C++ standard.

2 For any given type T, the result of applying one of these templates to T and to cv T shall yield the same result.

3 [Note 1: For any given type T, exactly one of the primary type categories has a value member that evaluates to true. —end note]

Table 47: Primary type category predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct is_void;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_null_pointer;</td>
<td>T is nullptr_t (6.8.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_integral;</td>
<td>T is an integral type (6.8.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_floating_point;</td>
<td>T is a floating-point type (6.8.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_array;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_pointer;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_lvalue_reference;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_rvalue_reference;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_member_object_pointer;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_member_function_pointer;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_enum;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_union;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_class;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_function;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20.15.5.3 Composite type traits

1 These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in subclause 6.8.

2 For any given type T, the result of applying one of these templates to T and to cv T shall yield the same result.

Table 48: Composite type category predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct is_reference;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_arithmetic;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_fundamental;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 48: Composite type category predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct is_object;</td>
<td>T is an object type (6.8)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_scalar;</td>
<td>T is a scalar type (6.8)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_compound;</td>
<td>T is a compound type (6.8.3)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_member_pointer;</td>
<td>T is a pointer-to-member type (6.8.3)</td>
<td></td>
</tr>
</tbody>
</table>

20.15.5.4 Type properties [meta.unary.prop]

1 These templates provide access to some of the more important properties of types.
2 It is unspecified whether the library defines any full or partial specializations of any of these templates.
3 For all of the class templates X declared in this subclause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of X require that the argument is a complete type.
4 For the purpose of defining the templates in this subclause, a function call expression declval<T>() for any type T is considered to be a trivial (6.8, 11.4.4) function call that is not an odr-use (6.3) of declval in the context of the corresponding definition notwithstanding the restrictions of 20.2.6.

Table 49: Type property predicates [tab:meta.unary.prop]

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct is_const;</td>
<td>T is const-qualified (6.8.4)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_volatile;</td>
<td>T is volatile-qualified (6.8.4)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivial;</td>
<td>T is a trivial type (6.8) remove_all_extents_-t&lt;T&gt; shall be a complete type or cv void.</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivially_copyable;</td>
<td>T is a trivially copyable type (6.8) remove_all_extents_-t&lt;T&gt; shall be a complete type or cv void.</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_standard_layout;</td>
<td>T is a standard-layout type (6.8) remove_all_extents_-t&lt;T&gt; shall be a complete type or cv void.</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_empty;</td>
<td>T is a class type, but not a union type, with no non-static data members other than subobjects of zero size, no virtual member functions, no virtual base classes, and no base class B for which is_empty_v&lt;B&gt; is false. If T is a non-union class type, T shall be a complete type.</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_polymorphic;</td>
<td>T is a polymorphic class (11.7.3) If T is a non-union class type, T shall be a complete type.</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_abstract;</td>
<td>T is an abstract class (11.7.4) If T is a non-union class type, T shall be a complete type.</td>
<td></td>
</tr>
<tr>
<td>Template</td>
<td>Condition</td>
<td>Preconditions</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_final;</td>
<td>T is a class type marked with the <code>class-virt-specifier</code> <code>final</code> (11.1). [Note 1: A union is a class type that can be marked with <code>final</code>. — end note]</td>
<td>If T is a class type, T shall be a complete type.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_aggregate;</td>
<td>T is an aggregate type (9.4.2) remove_all_extents_t&lt;T&gt; shall be a complete type or <code>cv void</code>.</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_signed;</td>
<td>If <code>is_arithmetic_v&lt;T&gt;</code> is true, the same result as T(-1) &lt; T(0); otherwise, false</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_unsigned;</td>
<td>If <code>is_arithmetic_v&lt;T&gt;</code> is true, the same result as T(0) &lt; T(-1); otherwise, false</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_bounded_array;</td>
<td>T is an array type of known bound (9.3.4.5)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_unbounded_array;</td>
<td>T is an array type of unknown bound (9.3.4.5)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T, class... Args&gt; struct is_constructible;</td>
<td>For a function type T or for a <code>cv void</code> type T, <code>is_constructible_v&lt;T, Args...&gt;</code> is false, otherwise see below</td>
<td>T and all types in the template parameter pack Args shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_default_constructible;</td>
<td><code>is_constructible_v&lt;T&gt;</code> is true.</td>
<td>T shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_copy_constructible;</td>
<td>For a referenceable type T (3.45), the same result as <code>is_constructible_v&lt;T, const T&amp;&gt;</code>, otherwise false.</td>
<td>T shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_move_constructible;</td>
<td>For a referenceable type T, the same result as <code>is_constructible_v&lt;T, T&amp;&amp;&gt;</code>, otherwise false.</td>
<td>T shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 49: Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
</table>
| template<class T, class U> struct is_assignable; | The expression `declval<T>() = declval<U>()` is well-formed when treated as an unevaluated operand (7.2). Access checking is performed as if in a context unrelated to T and U. Only the validity of the immediate context of the assignment expression is considered.  
[Note 2: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. —end note] | T and U shall be complete types, cv void, or arrays of unknown bound. |
| template<class T> struct is_copy_assignable; | For a referenceable type T, the same result as `is_assignable_v<T&, const T&>`; otherwise false. | T shall be a complete type, cv void, or an array of unknown bound. |
| template<class T> struct is_move_assignable; | For a referenceable type T, the same result as `is_assignable_v<T&, T&&>`; otherwise false. | T shall be a complete type, cv void, or an array of unknown bound. |
Table 49: Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T, class U&gt; struct is_swappable_with;</td>
<td>The expressions <code>swap(declval&lt;T&gt;(), declval&lt;U&gt;())</code> and <code>swap(declval&lt;U&gt;(), declval&lt;T&gt;())</code> are each well-formed when treated as an unevaluated operand (7.2) in an overload-resolution context for swappable values (16.4.4.3). Access checking is performed as if in a context unrelated to T and U. Only the validity of the immediate context of the <code>swap</code> expressions is considered. [Note 3: The compilation of the expressions can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. —end note]</td>
<td>T and U shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_swappable;</td>
<td>For a referenceable type T, the same result as <code>is_swappable_with_v&lt;T&amp;, T&amp;&gt;</code>, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_destructible;</td>
<td>Either T is a reference type, or T is a complete object type for which the expression <code>declval&lt;U&amp;&gt;()</code> is well-formed when treated as an unevaluated operand (7.2), where U is <code>remove_all_extents_t&lt;T&gt;</code>.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class... Args&gt; struct is_trivially_constructible;</td>
<td><code>is_constructible_v&lt;T, Args...&gt;</code> is true and the variable definition for <code>is_constructible</code>, as defined below, is known to call no operation that is not trivial (6.8, 11.4.4).</td>
<td>T and all types in the template parameter pack Args shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivially_default_constructible;</td>
<td>`is_trivially_constructible_v&lt;T&gt; is true.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
## Table 49: Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct is_trivially_copy_constructible;</td>
<td>For a referenceable type ( T ), the same result as ( \text{is_trivially_constructible_v}&lt;T, \text{const } T&amp;&gt; ), otherwise false.</td>
<td>( T ) shall be a complete type, ( \text{cv } \text{void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivially_move_constructible;</td>
<td>For a referenceable type ( T ), the same result as ( \text{is_trivially_constructible_v}&lt;T, T&amp;&amp;&gt; ), otherwise false.</td>
<td>( T ) shall be a complete type, ( \text{cv } \text{void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class U&gt; struct is_trivially_assignable;</td>
<td>( \text{is_assignable_v}&lt;T, U&gt; ) is true and the assignment, as defined by ( \text{is_assignable} ), is known to call no operation that is not trivial (6.8, 11.4.4).</td>
<td>( T ) and ( U ) shall be complete types, ( \text{cv } \text{void} ), or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivially_copy_assignable;</td>
<td>For a referenceable type ( T ), the same result as ( \text{is_trivially_assignable_v}&lt;T&amp;, \text{const } T&amp;&gt; ), otherwise false.</td>
<td>( T ) shall be a complete type, ( \text{cv } \text{void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivially_move_assignable;</td>
<td>For a referenceable type ( T ), the same result as ( \text{is_trivially_assignable_v}&lt;T&amp;, T&amp;&amp;&gt; ), otherwise false.</td>
<td>( T ) shall be a complete type, ( \text{cv } \text{void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivially_destructible;</td>
<td>( \text{is_destructible_v}&lt;T&gt; ) is true and ( \text{remove_all_extents_t}&lt;T&gt; ) is either a non-class type or a class type with a trivial destructor.</td>
<td>( T ) shall be a complete type, ( \text{cv } \text{void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class... Args&gt; struct is_nothrow_constructible;</td>
<td>( \text{is_constructible_v}&lt;T, \text{Args...}&gt; ) is true and the variable definition for ( \text{is_constructible} ), as defined below, is known not to throw any exceptions (7.6.2.7).</td>
<td>( T ) and all types in the template parameter pack ( \text{Args} ) shall be complete types, ( \text{cv } \text{void} ), or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_default_constructible;</td>
<td>( \text{is_nothrow_constructible_v}&lt;T&gt; ) is true.</td>
<td>( T ) shall be a complete type, ( \text{cv } \text{void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_copy_constructible;</td>
<td>For a referenceable type ( T ), the same result as ( \text{is_nothrow_constructible_v}&lt;T, \text{const } T&amp;&gt; ), otherwise false.</td>
<td>( T ) shall be a complete type, ( \text{cv } \text{void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_move_constructible;</td>
<td>For a referenceable type ( T ), the same result as ( \text{is_nothrow_constructible_v}&lt;T, T&amp;&amp;&gt; ), otherwise false.</td>
<td>( T ) shall be a complete type, ( \text{cv } \text{void} ), or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 49: Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T, class U&gt; struct is_nothrow_assignable;</td>
<td>is_assignable_v&lt;T, U&gt; is true and the assignment is known not to throw any exceptions (7.6.2.7).</td>
<td>T and U shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_copy_assignable;</td>
<td>For a referenceable type T, the same result as is_nothrow_assignable_v&lt;T&amp;, const T&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_move_assignable;</td>
<td>For a referenceable type T, the same result as is_nothrow_assignable_v&lt;T&amp;, T&amp;&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class U&gt; struct is_nothrow_swappable_with;</td>
<td>is_swappable_with_v&lt;T, U&gt; is true and each swap expression of the definition of is_swappable_with&lt;T, U&gt; is known not to throw any exceptions (7.6.2.7).</td>
<td>T and U shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_swappable;</td>
<td>For a referenceable type T, the same result as is_nothrow_swappable_v&lt;T&amp;, T&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_destructible;</td>
<td>is_destructible_v&lt;T&gt; is true and the indicated destructor is known not to throw any exceptions (7.6.2.7).</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct has_virtualDestructor;</td>
<td>T has a virtual destructor (11.4.7)</td>
<td>If T is a non-union class type, T shall be a complete type.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct has_unique_object_representations;</td>
<td>For an array type T, the same result as has_unique_object_representations_v&lt;remove_all_extents_v&lt;remove_all_extents_v&lt;T&gt;&gt;&gt;, otherwise see below.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
</tbody>
</table>

5 [Example 1:]

is_const_v<const volatile int> // true
is_const_v<const int*> // false
is_const_v<const int&> // false
is_const_v<int[3]> // false
is_const_v<const int[3]> // true

— end example []

6 [Example 2:]

remove_const_t<const volatile int> // volatile int
remove_const_t<const int* const> // const int*
remove_const_t<const int&> // const int&

— end example []
Example 3:

// Given:
struct P final { };
union U1 { };
union U2 final { };

// the following assertions hold:
static_assert(!is_final_v<int>);
static_assert(is_final_v<P>);
static_assert(!is_final_v<U1>);
static_assert(is_final_v<U2>);
— end example

The predicate condition for a template specialization is_constructible<T, Args...> shall be satisfied if and only if the following variable definition would be well-formed for some invented variable t:

T t(declval<Args>()...);

Note 4: These tokens are never interpreted as a function declaration. — end note

Access checking is performed as if in a context unrelated to T and any of the Args. Only the validity of the immediate context of the variable initialization is considered.

Note 5: The evaluation of the initialization can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. — end note

The predicate condition for a template specialization has_unique_object_representations<T> shall be satisfied if and only if:

9 (9.1) — T is trivially copyable, and
(9.2) — any two objects of type T with the same value have the same object representation, where two objects of array or non-union class type are considered to have the same value if their respective sequences of direct subobjects have the same values, and two objects of union type are considered to have the same value if they have the same active member and the corresponding members have the same value.

The set of scalar types for which this condition holds is implementation-defined.

Note 6: If a type has padding bits, the condition does not hold; otherwise, the condition holds true for integral types. — end note

20.15.6 Type property queries [meta.unary.prop.query]

This subclause contains templates that may be used to query properties of types at compile time.

<table>
<thead>
<tr>
<th>Template</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct alignment_of;</td>
<td>alignof(T).</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct rank;</td>
<td>If T names an array type, an integer value representing the number of dimensions of T; otherwise, 0.</td>
</tr>
<tr>
<td>template&lt;class T, unsigned I = 0&gt; struct extent;</td>
<td>If T is not an array type, or if it has rank less than or equal to I, or if I is 0 and T has type “array of unknown bound of U”, then 0; otherwise, the bound (9.3.4.5) of the I(^{\text{th}}) dimension of T, where indexing of I is zero-based</td>
</tr>
</tbody>
</table>

Each of these templates shall be a Cpp17UnaryTypeTrait (20.15.2) with a base characteristic of integral_-constant<size_t, Value>.

[Example 1:

// the following assertions hold:
assert(rank_v<int> == 0);
assert(rank_v<int[2]> == 1);
assert(rank_v<int[]>[4]> == 2);]
Example 2:

// the following assertions hold:
assert(extent_v<int> == 0);
assert(extent_v<int[2]> == 2);
assert(extent_v<int[2][4]> == 2);
assert(extent_v<int[] [4]> == 0);
assert((extent_v<int, 1>) == 0);
assert((extent_v<int[2], 1>) == 0);
assert((extent_v<int[2][4], 1>) == 4);
assert((extent_v<int[], 1>) == 4);

20.15.7 Relationships between types

This subclause contains templates that may be used to query relationships between types at compile time.

Each of these templates shall be a `Cpp17BinaryTypeTrait` (20.15.2) with a base characteristic of `true_type` if the corresponding condition is true, otherwise `false_type`.

### Table 51: Type relationship predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;class T, class U&gt; struct is_same;</code></td>
<td>T and U name the same type with the same cv-qualifications</td>
<td>If Base and Derived are non-union class types and are not possibly cv-qualified versions of the same type, Derived shall be a complete type.</td>
</tr>
<tr>
<td><code>template&lt;class Base, class Derived&gt; struct is_base_of;</code></td>
<td>Base is a base class of Derived (11.7) without regard to cv-qualifiers or Base and Derived are not unions and name the same class type without regard to cv-qualifiers</td>
<td>If Base and Derived are non-union class types and are not possibly cv-qualified versions of the same type, Derived shall be a complete type.</td>
</tr>
<tr>
<td><code>template&lt;class From, class To&gt; struct is_convertible;</code></td>
<td>see below</td>
<td>From and To shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>template&lt;class From, class To&gt; struct is_nothrow_convertible;</code></td>
<td>is_convertible_v&lt;From, To&gt; is true and the conversion, as defined by is_convertible, is known not to throw any exceptions (7.6.2.7)</td>
<td>From and To shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>template&lt;class T, class U&gt; struct is_layout_compatible;</code></td>
<td>T and U are layout-compatible (6.8)</td>
<td>T and U shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>template&lt;class Base, class Derived&gt; struct is_pointer_interconvertible_base_of;</code></td>
<td>Derived is unambiguously derived from Base without regard to cv-qualifiers, and each object of type Derived is pointer-interconvertible (6.8.3) with its Base subobject, or Base and Derived are not unions and name the same class type without regard to cv-qualifiers.</td>
<td>If Base and Derived are non-union class types and are not (possibly cv-qualified versions of) the same type, Derived shall be a complete type.</td>
</tr>
</tbody>
</table>
### Table 51: Type relationship predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class Fn, class... ArgTypes&gt; struct is_invocable;</td>
<td>The expression (\text{INVOKE} (\text{declval&lt;Fn&gt;()}, \text{declval&lt;ArgTypes&gt;()}...)) is well-formed when treated as an unevaluated operand.</td>
<td>Fn and all types in the template parameter pack ArgTypes shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class R, class Fn, class... ArgTypes&gt; struct is_invocable_r;</td>
<td>The expression (\text{INVOKE}&lt;R&gt;(\text{declval&lt;Fn&gt;()}, \text{declval&lt;ArgTypes&gt;()}...)) is well-formed when treated as an unevaluated operand.</td>
<td>Fn, R, and all types in the template parameter pack ArgTypes shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class Fn, class... ArgTypes&gt; struct is_nothrow_invocable;</td>
<td>(\text{is_invocable_v&lt;Fn, ArgTypes...&gt; is true and the expression } \text{INVOKE} (\text{declval&lt;Fn&gt;()}, \text{declval&lt;ArgTypes&gt;()}...)) is known not to throw any exceptions (7.6.2.7).</td>
<td>Fn and all types in the template parameter pack ArgTypes shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class R, class Fn, class... ArgTypes&gt; struct is_nothrow_invocable_r;</td>
<td>(\text{is_invocable_v&lt;R, Fn, ArgTypes...&gt; is true and the expression } \text{INVOKE}&lt;R&gt;(\text{declval&lt;Fn&gt;()}, \text{declval&lt;ArgTypes&gt;()}...)) is known not to throw any exceptions (7.6.2.7).</td>
<td>Fn, R, and all types in the template parameter pack ArgTypes shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
</tbody>
</table>

3 For the purpose of defining the templates in this subclause, a function call expression \(\text{declval<T>()}\) for any type \(T\) is considered to be a trivial \(6.8, 11.4.4\) function call that is not an odr-use \(6.3\) of \(\text{declval}\) in the context of the corresponding definition notwithstanding the restrictions of 20.2.6.

4 **[Example 1]**:

```cpp
struct B {}; 
struct B1 : B {}; 
struct B2 : B {}; 
struct D : private B1, private B2 {}; 

is_base_of_v<B, D> // true 
is_base_of_v<const B, D> // true 
is_base_of_v<const B, const D> // true 
is_base_of_v<int, D> // false 
is_base_of_v<int, int> // false 
```

**end example**

5 The predicate condition for a template specialization \(\text{is_convertible<From, To>}\) shall be satisfied if and only if the return expression in the following code would be well-formed, including any implicit conversions to the return type of the function:

```cpp
To test() {
    return declval<From>();
}
```

**[Note 2]:** This requirement gives well-defined results for reference types, void types, array types, and function types. **— end note**

Access checking is performed in a context unrelated to To and From. Only the validity of the immediate context of the expression of the return statement \(8.7.4\) (including initialization of the returned object or reference) is considered.
Note 3: The initialization can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. —end note

20.15.8 Transformations between types [meta.trans]

20.15.8.1 General [meta.trans.general]

1 Subclause 20.15.8 contains templates that may be used to transform one type to another following some predefined rule.

2 Each of the templates in 20.15.8 shall be a \texttt{C++17TransformationTrait} (20.15.2).

20.15.8.2 Const-volatile modifications [meta.trans.cv]

Table 52: Const-volatile modifications [tab:meta.trans.cv]

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{template&lt;class T&gt; struct remove_const;}; }\texttt{struct remove_const;}</td>
<td>The member typedef \texttt{type} names the same type as \texttt{T} except that any top-level const-qualifier has been removed. \texttt{[Example 1: remove_const_t\langle\texttt{const volatile int}\rangle evaluates to volatile int, whereas remove_const_t\langle\texttt{const int*}\rangle evaluates to const int*. —end example]}</td>
</tr>
<tr>
<td>\texttt{template&lt;class T&gt; struct remove_volatile;}; }\texttt{struct remove_volatile;}</td>
<td>The member typedef \texttt{type} names the same type as \texttt{T} except that any top-level volatile-qualifier has been removed. \texttt{[Example 2: remove_volatile_t\langle\texttt{const volatile int}\rangle evaluates to const int, whereas remove_volatile_t\langle\texttt{volatile int*}\rangle evaluates to volatile int*. —end example]}</td>
</tr>
<tr>
<td>\texttt{template&lt;class T&gt; struct remove_cv;}; }\texttt{struct remove_cv;}</td>
<td>The member typedef \texttt{type} shall be the same as \texttt{T} except that any top-level cv-qualifier has been removed. \texttt{[Example 3: remove_cv_t\langle\texttt{const volatile int}\rangle evaluates to int, whereas remove_cv_t\langle\texttt{volatile int*}\rangle evaluates to const volatile int*. —end example]}</td>
</tr>
<tr>
<td>\texttt{template&lt;class T&gt; struct add_const;}; }\texttt{struct add_const;}</td>
<td>Type \texttt{T} is a reference, function, or top-level const-qualified type, then \texttt{type} names the same type as \texttt{T}, otherwise \texttt{T const}.</td>
</tr>
<tr>
<td>\texttt{template&lt;class T&gt; struct add_volatile;}; }\texttt{struct add_volatile;}</td>
<td>Type \texttt{T} is a reference, function, or top-level volatile-qualified type, then \texttt{type} names the same type as \texttt{T}, otherwise \texttt{T volatile}.</td>
</tr>
<tr>
<td>\texttt{template&lt;class T&gt; struct add_cv;}; }\texttt{struct add_cv;}</td>
<td>The member typedef \texttt{type} names the same type as \texttt{add_const_t\langle add_volatile_t\langle T\rangle\rangle}.</td>
</tr>
</tbody>
</table>

20.15.8.3 Reference modifications [meta.trans.ref]

Table 53: Reference modifications [tab:meta.trans.ref]

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{template&lt;class T&gt; struct remove_reference;}; }\texttt{struct remove_reference;}</td>
<td>If \texttt{T} has type “reference to \texttt{T1}” then the member typedef \texttt{type} names \texttt{T1}; otherwise, \texttt{type} names \texttt{T}.</td>
</tr>
<tr>
<td>\texttt{template&lt;class T&gt; struct add_lvalue_reference;}; }\texttt{struct add_lvalue_reference;}</td>
<td>If \texttt{T} names a referenceable type (3.45) then the member typedef \texttt{type} names \texttt{T&amp;}; otherwise, \texttt{type} names \texttt{T}. \texttt{[Note 1: This rule reflects the semantics of reference collapsing (9.3.4.3). —end note]}</td>
</tr>
<tr>
<td>\texttt{template&lt;class T&gt; struct add_rvalue_reference;}; }\texttt{struct add_rvalue_reference;}</td>
<td>If \texttt{T} names a referenceable type then the member typedef \texttt{type} names \texttt{T&amp;&amp;}; otherwise, \texttt{type} names \texttt{T}. \texttt{[Note 2: This rule reflects the semantics of reference collapsing (9.3.4.3). For example, when a type \texttt{T} names a type \texttt{T1&amp;}, the type \texttt{add_rvalue_reference_t\langle T\rangle} is not an rvalue reference. —end note]}</td>
</tr>
</tbody>
</table>

20.15.8.4 Sign modifications [meta.trans.sign]
Table 54: Sign modifications  [tab:meta.trans.sign]

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| template<class T> struct make_signed; | If \( T \) names a (possibly cv-qualified) signed integer type \( \text{(6.8.2)} \) then the member typedef \( \text{type} \) names the type \( T \); otherwise, if \( T \) names a (possibly cv-qualified) unsigned integer type then \( \text{type} \) names the corresponding signed integer type, with the same cv-qualifiers as \( T \); otherwise, \( \text{type} \) names the signed integer type with smallest rank \( \text{(6.8.5)} \) for which \( \text{sizeof}(T) = \text{sizeof(type)} \), with the same cv-qualifiers as \( T \). 
Mandates: \( T \) is an integral or enumeration type other than \( \text{cv bool} \). |
| template<class T> struct make_unsigned; | If \( T \) names a (possibly cv-qualified) unsigned integer type \( \text{(6.8.2)} \) then the member typedef \( \text{type} \) names the type \( T \); otherwise, if \( T \) names a (possibly cv-qualified) signed integer type then \( \text{type} \) names the corresponding unsigned integer type, with the same cv-qualifiers as \( T \); otherwise, \( \text{type} \) names the unsigned integer type with smallest rank \( \text{(6.8.5)} \) for which \( \text{sizeof}(T) = \text{sizeof(type)} \), with the same cv-qualifiers as \( T \). 
Mandates: \( T \) is an integral or enumeration type other than \( \text{cv bool} \). |

20.15.8.5 Array modifications  [meta.trans.arr]

Table 55: Array modifications  [tab:meta.trans.arr]

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| template<class T> struct remove_extent; | If \( T \) names a type “array of \( U \)”, the member typedef \( \text{type} \) shall be \( U \), otherwise \( T \). 
[Note 1: For multidimensional arrays, only the first array dimension is removed. For a type “array of const \( U \)”, the resulting type is const \( U \). —end note] |
| template<class T> struct remove_all_extents; | If \( T \) is “multi-dimensional array of \( U \)”, the resulting member typedef \( \text{type} \) is \( U \), otherwise \( T \). |

1 [Example 1:]
// the following assertions hold:
assert((is_same_v<remove_extent_t<int>, int>));
assert((is_same_v<remove_extent_t<int[2]>, int>));
assert((is_same_v<remove_extent_t<int[2][3]>, int[3]>));
assert((is_same_v<remove_extent_t<int[1][3]>, int[3]>));
—end example]

2 [Example 2:]
// the following assertions hold:
assert((is_same_v<remove_all_extents_t<int>, int>));
assert((is_same_v<remove_all_extents_t<int[2]>, int>));
assert((is_same_v<remove_all_extents_t<int[2][3]>, int>));
assert((is_same_v<remove_all_extents_t<int[1][3]>, int>));
—end example]

20.15.8.6 Pointer modifications  [meta.trans.ptr]

Table 56: Pointer modifications  [tab:meta.trans.ptr]

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct remove_pointer;</td>
<td>If ( T ) has type “(possibly cv-qualified) pointer to ( T1 )” then the member typedef ( \text{type} ) names ( T1 ); otherwise, it names ( T ).</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct add_pointer;</td>
<td>If ( T ) names a referenceable type ( \text{(3.45)} ) or a ( \text{cv void} ) type then the member typedef ( \text{type} ) names the same type as ( \text{remove_reference_t&lt;T&gt;}* ); otherwise, ( \text{type} ) names ( T ).</td>
</tr>
</tbody>
</table>
### 20.15.8.7 Other transformations

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;class T&gt; struct type_identity;</code></td>
<td>The member typedef <code>type</code> names the type <code>T</code>.</td>
</tr>
<tr>
<td><code>template&lt;size_t Len, size_t Align = default-alignment&gt; struct aligned_storage;</code></td>
<td>The value of <code>default-alignment</code> shall be the most stringent alignment requirement for any object type whose size is no greater than <code>Len</code> (6.8). The member typedef <code>type</code> shall be a trivial standard-layout type suitable for use as uninitialized storage for any object whose size is at most <code>Len</code> and whose alignment is a divisor of <code>Align</code>. <strong>Mandates:</strong> <code>Len</code> is not zero. <code>Align</code> is equal to <code>alignof(T)</code> for some type <code>T</code> or to <code>default-alignment</code>.</td>
</tr>
<tr>
<td><code>template&lt;size_t Len, class... Types&gt; struct aligned_union;</code></td>
<td>The member typedef <code>type</code> shall be a trivial standard-layout type suitable for use as uninitialized storage for any object whose type is listed in <code>Types</code>; its size shall be at least <code>Len</code>. The static member <code>alignment_value</code> shall be an integral constant of type <code>size_t</code> whose value is the strictest alignment of all types listed in <code>Types</code>. <strong>Mandates:</strong> At least one type is provided. Each type in the template parameter pack <code>Types</code> is a complete object type.</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct remove_cvref;</code></td>
<td>The member typedef <code>type</code> names the same type as <code>remove_cv_t&lt;remove_reference_t&lt;T&gt;&gt;</code>.</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct decay;</code></td>
<td>Let <code>U</code> be <code>remove_reference_t&lt;T&gt;</code>. If <code>is_array_v&lt;U&gt;</code> is true, the member typedef <code>type</code> equals <code>remove_extent_t&lt;U&gt;</code>. If <code>is_function_v&lt;U&gt;</code> is true, the member typedef <code>type</code> equals <code>add_pointer_t&lt;U&gt;</code>. Otherwise the member typedef <code>type</code> equals <code>remove_cv_t&lt;U&gt;</code>. <strong>[Note 1: This behavior is similar to the lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and function-to-pointer (7.3.4) conversions applied when an lvalue is used as an rvalue, but also strips cv-qualifiers from class types in order to more closely model by-value argument passing. —end note]</strong></td>
</tr>
<tr>
<td><code>template&lt;bool B, class T = void&gt; struct enable_if;</code></td>
<td>If <code>B</code> is true, the member typedef <code>type</code> shall equal <code>T</code>; otherwise, there shall be no member <code>type</code>.</td>
</tr>
<tr>
<td><code>template&lt;bool B, class T, class F&gt; struct conditional;</code></td>
<td>If <code>B</code> is true, the member typedef <code>type</code> shall equal <code>T</code>. If <code>B</code> is false, the member typedef <code>type</code> shall equal <code>F</code>.</td>
</tr>
<tr>
<td><code>template&lt;class... T&gt; struct common_type;</code></td>
<td>Unless this trait is specialized (as specified in Note B, below), the member <code>type</code> is defined or omitted as specified in Note A, below. If it is omitted, there shall be no member <code>type</code>. Each type in the template parameter pack <code>T</code> shall be complete, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template&lt;class, class, template&lt;class&gt; class, template&lt;class&gt; class&gt; struct basic_common_reference;</code></td>
<td>The member <code>typedef-name</code> <code>type</code> is defined or omitted as specified in Note C, below. Each type in the parameter pack <code>T</code> shall be complete or <code>cv void</code>.</td>
</tr>
<tr>
<td><code>template&lt;class... T&gt; struct common_reference;</code></td>
<td>If <code>T</code> is an enumeration type, the member typedef <code>type</code> names the underlying type of <code>T</code> (9.7.1); otherwise, there is no member <code>type</code>. <strong>Mandates:</strong> <code>T</code> is not an incomplete enumeration type.</td>
</tr>
</tbody>
</table>
Table 57: Other transformations (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class Fn,</td>
<td>If the expression <code>INVOKE</code>DECLVAL(Fn)(), DECLVAL&lt;ArgTypes&gt;()...) is well-formed when treated as an unvaluated operand (7.2), the member typedef type names the type decltype(INVOKE(DECLVAL(Fn)(), DECLVAL&lt;ArgTypes&gt;()...)); otherwise, there shall be no member type. Access checking is performed as if in a context unrelated to Fn and ArgTypes. Only the validity of the immediate context of the expression is considered.</td>
</tr>
<tr>
<td>class... ArgTypes&gt;</td>
<td></td>
</tr>
<tr>
<td>struct invoke_result;</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct</td>
<td>If T is a specialization reference_wrapper&lt;X&gt; for some type X, the member typedef type of unwrap_reference&lt;T&gt; is X&amp;, otherwise it is T.</td>
</tr>
<tr>
<td>unwrap_reference;</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>The member typedef type of unwrap_ref_decay&lt;T&gt; denotes the type unwrap_reference_t&lt;decay_t&lt;T&gt;&gt;.</td>
</tr>
<tr>
<td>unwrap_ref_decay;</td>
<td></td>
</tr>
</tbody>
</table>

1. [Note 3: A typical implementation would define aligned_storage as:

```cpp
template<size_t Len, size_t Alignment>
struct aligned_storage {
    typedef struct {
        alignas(Alignment) unsigned char _data[Len];
    } type;
};
```

--- end note

2. In addition to being available via inclusion of the <type_traits> header, the templates unwrap_reference, unwrap_ref_decay, unwrap_reference_t, and unwrap_ref_decay_t are available when the header <functional> (20.14.2) is included.

3. Let:

- CREF(A) be add_lvalue_reference_t<const remove_reference_t<A>>,
- XREF(A) denote a unary alias template T such that T<U> denotes the same type as U with the addition of A's cv and reference qualifiers, for a non-reference cv-unqualified type U,
- COPYCV(FROM, TO) be an alias for type TO with the addition of FROM's top-level cv-qualifiers,
- [Example 1: COPYCV(const int, volatile short) is an alias for const volatile short. — end example]
- COND-RES(X, Y) be decltype(false ? declval<X(&)()>()() : declval<Y(&)()>()())

Given types A and B, let X be remove_reference_t<A>, let Y be remove_reference_t<B>, and let COMMON-REF(A, B) be:

- If A and B are both lvalue reference types, COMMON-REF(A, B) is COND-RES(COPYCV(X, Y) &, COPYCV(Y, X) &)
- Otherwise, let C be remove_reference_t<COMMON-REF(X&, Y)&>&. If A and B are both rvalue reference types, C is well-formed, and is_convertible_v<C<A, C> && is_convertible_v<C,B, C> is true, then COMMON-REF(A, B) is C.
- Otherwise, let D be COMMON-REF(const X&, Y&). If A is an rvalue reference and B is an lvalue reference and D is well-formed and is_convertible_v<C,A> D> is true, then COMMON-REF(A, B) is D.
- Otherwise, if A is an lvalue reference and B is an rvalue reference, then COMMON-REF(A, B) is COMMON-REF(B, A).
- Otherwise, COMMON-REF(A, B) is ill-formed.

§ 20.15.8.7 729
If any of the types computed above is ill-formed, then \textit{COMMON-REF}(A, B) is ill-formed.

Note A: For the \textit{common_type} trait applied to a template parameter pack \(T\) of types, the member \textit{type} shall be either defined or not present as follows:

\begin{align*}
\text{(4.1)} & \quad \text{If } \text{sizeof} \ldots(T) \text{ is zero, there shall be no member type.} \\
\text{(4.2)} & \quad \text{If } \text{sizeof} \ldots(T) \text{ is one, let } T_0 \text{ denote the sole type constituting the pack } T. \text{ The member typedef-name type shall denote the same type, if any, as } \text{common_type}_t<T_0, T_0>; \text{ otherwise there shall be no member type.} \\
\text{(4.3)} & \quad \text{If } \text{sizeof} \ldots(T) \text{ is two, let the first and second types constituting } T \text{ be denoted by } T_1 \text{ and } T_2, \text{ respectively, and let } D_1 \text{ and } D_2 \text{ denote the same types as decay_t}<T_1> \text{ and decay_t}<T_2>, \text{ respectively.} \\
\text{(4.3.1)} & \quad \text{If } \text{is_same_v}<T_1, D_1> \text{ is false or } \text{is_same_v}<T_2, D_2> \text{ is false, let } C \text{ denote the same type, if any, as common_type_t}<D_1, D_2>. \\
\text{(4.3.2)} & \quad \text{[Note 4: None of the following will apply if there is a specialization common_type_t}<D_1, D_2>. \text{ — end note]} \\
\text{(4.3.3)} & \quad \text{Otherwise, if } \\
\text{decay_t}<\text{decltype}(\text{false ? declval<D1>() : declval<D2>())> denotes a valid type, let } C \text{ denote that type.} \\
\text{(4.3.4)} & \quad \text{Otherwise, if } \text{COND-RES}(\text{CREF}(D_1), \text{CREF}(D_2)) \text{ denotes a type, let } C \text{ denote the type } \text{decay_t}<\text{COND-RES}(\text{CREF}(D_1), \text{CREF}(D_2)))>. \\
\text{In either case, the member typedef-name type shall denote the same type, if any, as } C. \text{ Otherwise, there shall be no member type.} \\
\text{(4.4)} & \quad \text{If } \text{sizeof} \ldots(T) \text{ is greater than two, let } T_1, T_2, \text{ and } R, \text{ respectively, denote the first, second, and (pack of) remaining types constituting } T. \text{ Let } C \text{ denote the same type, if any, as common_type_t}<T_1, T_2>. \text{ If there is such a type } C, \text{ the member typedef-name type shall denote the same type, if any, as common_type_t}<T_2, C, R \ldots>. \text{ Otherwise, there shall be no member type.} \\
\end{align*}

Note B: Notwithstanding the provisions of 20.15.3, and pursuant to 16.4.5.2.1, a program may specialize common_type_t\(<T_1, T_2>\) for types \(T_1\) and \(T_2\) such that is_same_v\(<T_1, decay_t<T_1>>\) and is_same_v\(<T_2, decay_t<T_2>>\) are each true.

[Note 5: Such specializations are needed when only explicit conversions are desired between the template arguments. — end note]

Such a specialization need not have a member named type, but if it does, that member shall be a typedef-name for an accessible and unambiguous cv-unqualified non-reference type \(C\) to which each of the types \(T_1\) and \(T_2\) is explicitly convertible. Moreover, common_type_t\(<T_1, T_2>\) shall denote the same type, if any, as does common_type_t\(<T_2, T_1>\). No diagnostic is required for a violation of this Note’s rules.

Note C: For the \textit{common_reference} trait applied to a parameter pack \(T\) of types, the member \textit{type} shall be either defined or not present as follows:

\begin{align*}
\text{(6.1)} & \quad \text{If } \text{sizeof} \ldots(T) \text{ is zero, there shall be no member type.} \\
\text{(6.2)} & \quad \text{Otherwise, if } \text{sizeof} \ldots(T) \text{ is one, let } T_0 \text{ denote the sole type in the pack } T. \text{ The member typedef type shall denote the same type as } T_0. \\
\text{(6.3)} & \quad \text{Otherwise, if } \text{sizeof} \ldots(T) \text{ is two, let } T_1 \text{ and } T_2 \text{ denote the two types in the pack } T. \text{ Then} \\
\text{(6.3.1)} & \quad \text{If } T_1 \text{ and } T_2 \text{ are reference types and } \text{COMMON-REF}(T_1, T_2) \text{ is well-formed, then the member typedef type denotes that type.} \\
\text{(6.3.2)} & \quad \text{Otherwise, if } \text{basic_common_reference}<\text{remove_cvref_t}<T_1>, \text{remove_cvref_t}<T_2>, \text{XREF}(T_1), \text{XREF}(T_2)>::\text{type} \text{ is well-formed, then the member typedef type denotes that type.} \\
\text{(6.3.3)} & \quad \text{Otherwise, if } \text{COND-RES}(T_1, T_2) \text{ is well-formed, then the member typedef type denotes that type.} \\
\text{(6.3.4)} & \quad \text{Otherwise, if } \text{common_type_t}<T_1, T_2> \text{ is well-formed, then the member typedef type denotes that type.} \\
\text{(6.3.5)} & \quad \text{Otherwise, there shall be no member type.} \\
\text{(6.4)} & \quad \text{Otherwise, if } \text{sizeof} \ldots(T) \text{ is greater than two, let } T_1, T_2, \text{ and } \text{Rest}, \text{ respectively, denote the first, second, and (pack of) remaining types comprising } T. \text{ Let } C \text{ be the type } \text{common_reference_t}<T_1, T_2>. \text{ Then:} \\
\end{align*}
— If there is such a type C, the member typedef type shall denote the same type, if any, as common_reference_t<C, Rest...>.

— Otherwise, there shall be no member type.

Note D: Notwithstanding the provisions of 20.15.3, and pursuant to 16.4.5.2.1, a program may partially specialize basic_common_reference<T, U, TQual, UQual> for types T and U such that is_same_v<T, decay_t<T>> and is_same_v<U, decay_t<U>> are each true.

[Note 6: Such specializations can be used to influence the result of common_reference, and are needed when only explicit conversions are desired between the template arguments. — end note]

Such a specialization need not have a member named type, but if it does, that member shall be a typedef-name for an accessible and unambiguous type C to which each of the types TQual<T> and UQual<U> is convertible. Moreover, basic_common_reference<T, U, TQual, UQual>::type shall denote the same type, if any, as does basic_common_reference<U, T, UQual, TQual>::type. No diagnostic is required for a violation of these rules.

Example 2: Given these definitions:

```cpp
using PF1 = bool (&)();
using PF2 = short (*)(long);
struct S {
    operator PF2() const;
    double operator() (char, int&);
    void fn(long) const;
    char data;
};
using PMF = void (S::*)(long) const;
using PMD = char S::*;
```

the following assertions will hold:

```cpp
static_assert(is_same_v<invoke_result_t<S, int>, short>);
static_assert(is_same_v<invoke_result_t<S&, unsigned char, int&>, double>);
static_assert(is_same_v<invoke_result_t<PF1, bool>);   
static_assert(is_same_v<invoke_result_t<PMF, unique_ptr<S>, int>, void>);
static_assert(is_same_v<invoke_result_t<PMF, S*, char&>>);   
static_assert(is_same_v<invoke_result_t<PMD, const S*, const char&>>);   
```

— end example]

20.15.9 Logical operator traits

This subclause describes type traits for applying logical operators to other type traits.

```cpp
template<class... B> struct conjunction : see below { 
    // The class template conjunction forms the logical conjunction of its template type arguments.
    // For a specialization conjunction<B1, ..., BN>, if there is a template type argument Bi for which
    // bool(Bi::value) is false, then instantiating conjunction<B1, ..., BN>::value does not require
    // the instantiation of B j::value for j > i.
    // [Note 1: This is analogous to the short-circuiting behavior of the built-in operator &&. — end note]
    // Every template type argument for which Bi::value is instantiated shall be usable as a base class and
    // shall have a member value which is convertible to bool, is not hidden, and is unambiguously available
    // in the type.
    // The specialization conjunction<B1, ..., BN> has a public and unambiguous base that is either
    // — the first type Bi in the list true_type, B1, ..., BN for which bool(Bi::value) is false, or
    // — if there is no such Bi, the last type in the list.
    // [Note 2: This means a specialization of conjunction does not necessarily inherit from either true_type or
    // false_type. — end note]
    // The member names of the base class, other than conjunction and operator*, shall not be hidden
    // and shall be unambiguously available in conjunction.

    // § 20.15.9
```
template<class... B> struct disjunction : see below {};  

The class template disjunction forms the logical disjunction of its template type arguments.

For a specialization disjunction<B₁, ..., Bᴺ>, if there is a template type argument Bᵢ for which bool(Bᵢ::value) is true, then instantiating disjunction<B₁, ..., Bᴺ>::value does not require the instantiation of Bⱼ::value for j > i.

[Note 3: This is analogous to the short-circuiting behavior of the built-in operator |. —end note]

Every template type argument for which Bᵢ::value is instantiated shall be usable as a base class and shall have a member value which is convertible to bool, is not hidden, and is unambiguously available in the type.

The specialization disjunction<B₁, ..., Bᴺ> has a public and unambiguous base that is either

— the first type Bᵢ in the list false_type, B₁, ..., Bᴺ for which bool(Bᵢ::value) is true, or

— if there is no such Bᵢ, the last type in the list.

[Note 4: This means a specialization of disjunction does not necessarily inherit from either true_type or false_type. —end note]

The member names of the base class, other than disjunction and operator=, shall not be hidden and shall be unambiguously available in disjunction.

template<class B> struct negation : see below {};  

The class template negation forms the logical negation of its template type argument. The type negation<B> is a Cpp17UnaryTypeTrait with a base characteristic of bool_constant<!bool(B::value)>.

20.15.10 Member relationships          [meta.member]

template<class S, class M>  
constexpr bool is_pointer_interconvertible_with_class(M S::*m) noexcept;  
1 Mandates: S is a complete type.
2 Returns: true if and only if S is a standard-layout type, M is an object type, m is not null, and each object s of type S is pointer-interconvertible (6.8.3) with its subobject s.*m.

template<class S1, class S2, class M1, class M2>  
constexpr bool is_corresponding_member(M1 S1::*m1, M2 S2::*m2) noexcept;  
3 Mandates: S1 and S2 are complete types.
4 Returns: true if and only if S1 and S2 are standard-layout types, M1 and M2 are object types, m1 and m2 are not null, and m1 and m2 point to corresponding members of the common initial sequence (11.4) of S1 and S2.

[Note 1: The type of a pointer-to-member expression &C::b is not always a pointer to member of C, leading to potentially surprising results when using these functions in conjunction with inheritance.

[Example 1]:
struct A { int a; }; // a standard-layout class
struct B { int b; }; // a standard-layout class
struct C: public A, public B { }; // not a standard-layout class

static_assert( is_pointer_interconvertible_with_class( &C::b ) );  // Succeeds because, despite its appearance, &C::b has type
static_assert( is_pointer_interconvertible_with_class< C >( &C::b ) );  // "pointer to member of B of type int".

static_assert( is_corresponding_member( &C::a, &C::b ) );  // Succeeds because, despite its appearance, &C::a and &C::b have types
static_assert( is_corresponding_member< C >( &C::a, &C::b ) );  // "pointer to member of A of type int" and
static_assert( is_corresponding_member< B >( &C::a, &C::b ) );  // "pointer to member of B of type int", respectively.

static_assert( is_corresponding_member< C, C >( &C::a, &C::b ) );  // Forces the use of class C, and fails.

static_assert( is_corresponding_member< &C::a, &C::b );  // Succeeds because, despite its appearance, &C::a and &C::b have types
static_assert( is_corresponding_member< &C::a, &C::b );  // "pointer to member of A of type int" and
static_assert( is_corresponding_member< &C::a, &C::b );  // "pointer to member of B of type int", respectively.

static_assert( is_corresponding_member< C, C >( &C::a, &C::b ) );  // Forces the use of class C, and fails.
20.15.11 Constant evaluation context

constexpr bool is_constant_evaluated() noexcept;

Returns: true if and only if evaluation of the call occurs within the evaluation of an expression or conversion that is manifestly constant-evaluated (7.7).

[Example 1:

corexpr void f(unsigned char *p, int n) {
    if (std::is_constant_evaluated()) {
        // should not be a corexpr if statement
        for (int k = 0; k<n; ++k) p[k] = 0;
    } else {
        memset(p, 0, n);
        // not a core constant expression
    }
}
—end example]
// 20.16.6, convenience SI typedefs
using yocto = ratio<1, 1'000'000'000'000'000'000'000'000>; // see below
using zepto = ratio<1, 1'000'000'000'000'000'000'000'000'>; // see below
using atto = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using femto = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using pico = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using nano = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using micro = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using milli = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using centi = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using deci = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using deca = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using hecto = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using kilo = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using mega = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using giga = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using tera = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using peta = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using exa = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using zetta = ratio<1, 1'000'000'000'000'000'000'000'000'>;
using yotta = ratio<1, 1'000'000'000'000'000'000'000'000'>;

20.16.3 Class template ratio
namespace std {
template<intmax_t N, intmax_t D = 1> class ratio {
public:
static constexpr intmax_t num;
static constexpr intmax_t den;
using type = ratio<num, den>;
};
}

1 If the template argument D is zero or the absolute values of either of the template arguments N and D is not representable by type intmax_t, the program is ill-formed.

[Note 1: These rules ensure that infinite ratios are avoided and that for any negative input, there exists a representable value of its absolute value which is positive. This excludes the most negative value. — end note]

2 The static data members num and den shall have the following values, where gcd represents the greatest common divisor of the absolute values of N and D:

(2.1) num shall have the value sign(N) * sign(D) * abs(N) / gcd.

(2.2) den shall have the value abs(D) / gcd.

20.16.4 Arithmetic on ratios
Each of the alias templates ratio_add, ratio_subtract, ratio_multiply, and ratio_divide denotes the result of an arithmetic computation on two ratios R1 and R2. With X and Y computed (in the absence of arithmetic overflow) as specified by Table 58, each alias denotes a ratio<U, V> such that U is the same as ratio<X, Y>::num and V is the same as ratio<X, Y>::den.

2 If it is not possible to represent U or V with intmax_t, the program is ill-formed. Otherwise, an implementation should yield correct values of U and V. If it is not possible to represent X or Y with intmax_t, the program is ill-formed unless the implementation yields correct values of U and V.

[Example 1:
static_assert(ratio_add<ratio<1, 3>, ratio<1, 6>>::num == 1, "1/3+1/6 == 1/2");
static_assert(ratio_add<ratio<1, 3>, ratio<1, 6>>::den == 2, "1/3+1/6 == 1/2");
static_assert(ratio_multiply<ratio<1, 3>, ratio<3, 2>>::num == 1, "1/3*3/2 == 1/2");
static_assert(ratio_multiply<ratio<1, 3>, ratio<3, 2>>::den == 2, "1/3*3/2 == 1/2");

// The following cases may cause the program to be ill-formed under some implementations
static_assert(ratio_add<ratio<1, INT_MAX>, ratio<1, INT_MAX>>::num == 2, "1/Max+1/Max == 2/Max");]
### Table 58: Expressions used to perform ratio arithmetic

<table>
<thead>
<tr>
<th>Type</th>
<th>Value of X</th>
<th>Value of Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>ratio_add&lt;R1, R2&gt;</td>
<td>R1::num * R2::den + R1::den * R2::den</td>
<td>R2::num * R1::den</td>
</tr>
<tr>
<td>ratio_subtract&lt;R1, R2&gt;</td>
<td>R1::num * R2::den - R1::den * R2::den</td>
<td>R2::num * R1::den</td>
</tr>
<tr>
<td>ratio_multiply&lt;R1, R2&gt;</td>
<td>R1::num * R2::num</td>
<td>R1::den * R2::den</td>
</tr>
<tr>
<td>ratio_divide&lt;R1, R2&gt;</td>
<td>R1::num * R2::num</td>
<td>R1::den * R2::den</td>
</tr>
</tbody>
</table>

static_assert(ratio_add<ratio<1, INT_MAX>, ratio<1, INT_MAX>>::den == INT_MAX, "1/MAX+1/MAX == 2/MAX");
static_assert(ratio_multiply<ratio<1, INT_MAX>, ratio<INT_MAX, 2>>::num == 1, "1/MAX * MAX/2 == 1/2");
static_assert(ratio_multiply<ratio<1, INT_MAX>, ratio<INT_MAX, 2>>::den == 2, "1/MAX * MAX/2 == 1/2");

—end example—

### 20.16.5 Comparison of ratios

```cpp
template<class R1, class R2>
struct ratio_equal : bool_constant<R1::num == R2::num && R1::den == R2::den> { }

template<class R1, class R2>
struct ratio_not_equal : bool_constant<!ratio_equal_v<R1, R2>> { }

template<class R1, class R2>
struct ratio_less : bool_constant<see below> { }

1 If R1::num × R2::den is less than R2::num × R1::den, ratio_less<R1, R2> shall be derived from
   bool_constant<true>; otherwise it shall be derived from bool_constant<false>. Implementations
   may use other algorithms to compute this relationship to avoid overflow. If overflow occurs, the program
   is ill-formed.

template<class R1, class R2>
struct ratio_less_equal : bool_constant<!ratio_less_v<R2, R1>> { }

template<class R1, class R2>
struct ratio_greater : bool_constant<ratio_less_v<R2, R1>> { }

template<class R1, class R2>
struct ratio_greater_equal : bool_constant<!ratio_less_v<R1, R2>> { }
```

### 20.16.6 SI types for ratio

1 For each of the typedef-names yocto, zepto, zetta, and yotta, if both of the constants used in its specification
   are representable by intmax_t, the typedef is defined; if either of the constants is not representable by
   intmax_t, the typedef is not defined.

### 20.17 Class type_index

#### 20.17.1 Header <typeindex> synopsis

```cpp
#include <compare> // see 17.11.1

namespace std {
  class type_index;
  template<class T> struct hash;
  template<> struct hash<type_index>;
}
```

§ 20.17 735
type_index overview

namespace std {
    class type_index {
    public:
        type_index(const type_info& rhs) noexcept;  
        bool operator==(const type_index& rhs) const noexcept;  
        bool operator<(const type_index& rhs) const noexcept;  
        bool operator>(const type_index& rhs) const noexcept;  
        bool operator<=(const type_index& rhs) const noexcept;  
        bool operator>=(const type_index& rhs) const noexcept;  
        strong_ordering operator<=>(const type_index& rhs) const noexcept;  
        size_t hash_code() const noexcept;  
        const char* name() const noexcept;  
    private:
        const type_info* target;  // exposition only
        // Note that the use of a pointer here, rather than a reference,
        // means that the default copy/move constructor and assignment
        // operators will be provided and work as expected.
    }
}

The class type_index provides a simple wrapper for type_info which can be used as an index type in associative containers (22.4) and in unordered associative containers (22.5).

type_index members

Effects: Constructs a type_index object, the equivalent of target = &rhs.

bool operator==(const type_index& rhs) const noexcept;

Returns: *target == *rhs.target.

bool operator<(const type_index& rhs) const noexcept;

Returns: target->before(*rhs.target).

bool operator>(const type_index& rhs) const noexcept;

Returns: rhs.target->before(*target).

bool operator<=(const type_index& rhs) const noexcept;

Returns: !rhs.target->before(*target).

bool operator>=(const type_index& rhs) const noexcept;

Returns: !target->before(*rhs.target).

strong_ordering operator<=>(const type_index& rhs) const noexcept;

Effects: Equivalent to:

if (*target == *rhs.target) return strong_ordering::equal;
if (target->before(*rhs.target)) return strong_ordering::less;
return strong_ordering::greater;

size_t hash_code() const noexcept;

Returns: target->hash_code().

const char* name() const noexcept;

Returns: target->name().
20.17.4 Hash support

```
template<> struct hash<type_index>;
```

For an object `index` of type `type_index`, `hash<type_index>()(index)` shall evaluate to the same result as `index.hash_code()`.

20.18 Execution policies

20.18.1 In general

Subclause 20.18 describes classes that are `execution policy` types. An object of an execution policy type indicates the kinds of parallelism allowed in the execution of an algorithm and expresses the consequent requirements on the element access functions.

Example 1:
```c++
using namespace std;
vector<int> v = /* ... */;

// standard sequential sort
sort(v.begin(), v.end());

// explicitly sequential sort
sort(execution::seq, v.begin(), v.end());

// permitting parallel execution
sort(execution::par, v.begin(), v.end());

// permitting vectorization as well
sort(execution::par_unseq, v.begin(), v.end());
```

—end example

Note 1: Because different parallel architectures might require idiosyncratic parameters for efficient execution, implementations can provide additional execution policies to those described in this standard as extensions. —end note

20.18.2 Header `<execution>` synopsis

```
namespace std {
    // 20.18.3, execution policy type trait
    template<class T> struct is_execution_policy;
    template<class T> inline constexpr bool is_execution_policy_v = is_execution_policy<T>::value;
}

namespace std::execution {
    // 20.18.4, sequenced execution policy
    class sequenced_policy;
    // 20.18.5, parallel execution policy
    class parallel_policy;
    // 20.18.6, parallel and unsequenced execution policy
    class parallel_unsequenced_policy;
    // 20.18.7, unsequenced execution policy
    class unsequenced_policy;
    // 20.18.8, execution policy objects
    inline constexpr sequenced_policy seq{ unspecified };
    inline constexpr parallel_policy par{ unspecified };
    inline constexpr parallel_unsequenced_policy par_unseq{ unspecified };
    inline constexpr unsequenced_policy unseq{ unspecified };
}
```
20.18.3 Execution policy type trait

\[
\text{template<class } T \text{> struct is_execution_policy { see below };}
\]

\text{\texttt{is\_execution\_policy} can be used to detect execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.}

\text{\texttt{is\_execution\_policy<T> is a \texttt{Cpp17UnaryTypeTrait} with a base characteristic of \texttt{true\_type} if } T \text{ is the type of a standard or implementation-defined execution policy, otherwise \texttt{false\_type}.}

\text{[Note 1: This provision reserves the privilege of creating non-standard execution policies to the library implementation. — end note]}

\text{The behavior of a program that adds specializations for \texttt{is\_execution\_policy} is undefined.}

20.18.4 Sequenced execution policy

\[
\text{class execution::sequenced\_policy \{ unspecified \};}
\]

\text{The class \texttt{execution::sequenced\_policy} is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized.}

\text{During the execution of a parallel algorithm with the \texttt{execution::sequenced\_policy} policy, if the invocation of an element access function exits via an uncaught exception, \texttt{terminate()} is called.}

20.18.5 Parallel execution policy

\[
\text{class execution::parallel\_policy \{ unspecified \};}
\]

\text{The class \texttt{execution::parallel\_policy} is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.}

\text{During the execution of a parallel algorithm with the \texttt{execution::parallel\_policy} policy, if the invocation of an element access function exits via an uncaught exception, \texttt{terminate()} is called.}

20.18.6 Parallel and unsequenced execution policy

\[
\text{class execution::parallel\_unsequenced\_policy \{ unspecified \};}
\]

\text{The class \texttt{execution::parallel\_unsequenced\_policy} is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized and vectorized.}

\text{During the execution of a parallel algorithm with the \texttt{execution::parallel\_unsequenced\_policy} policy, if the invocation of an element access function exits via an uncaught exception, \texttt{terminate()} is called.}

20.18.7 Unsequenced execution policy

\[
\text{class execution::unsequenced\_policy \{ unspecified \};}
\]

\text{The class \texttt{unsequenced\_policy} is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized, e.g., executed on a single thread using instructions that operate on multiple data items.}

\text{During the execution of a parallel algorithm with the \texttt{execution::unsequenced\_policy} policy, if the invocation of an element access function exits via an uncaught exception, \texttt{terminate()} is called.}

20.18.8 Execution policy objects

\[
\text{inline constexpr execution::sequenced\_policy execution::\texttt{seq}( unspecified );}
\]

\[
\text{inline constexpr execution::parallel\_policy execution::\texttt{par}( unspecified );}
\]

\[
\text{inline constexpr execution::parallel\_unsequenced\_policy execution::\texttt{par\_unseq}( unspecified );}
\]

\[
\text{inline constexpr execution::unsequenced\_policy execution::\texttt{unseq}( unspecified );}
\]

\text{The header \texttt{<execution>} declares global objects associated with each type of execution policy.}
20.19 Primitive numeric conversions

20.19.1 Header `<charconv>` synopsis

namespace std {
    // floating-point format for primitive numerical conversion
    enum class chars_format {
        scientific = unspecified,
        fixed = unspecified,
        hex = unspecified,
        general = fixed | scientific
    };

    // 20.19.2, primitive numerical output conversion
    struct to_chars_result {
        char* ptr;
        errc ec;
        friend bool operator==(const to_chars_result&, const to_chars_result&) = default;
    };

    to_chars_result to_chars(char* first, char* last, see below value, int base = 10);
    to_chars_result to_chars(char* first, char* last, bool value, int base = 10) = delete;

    to_chars_result to_chars(char* first, char* last, float value);
    to_chars_result to_chars(char* first, char* last, double value);
    to_chars_result to_chars(char* first, char* last, long double value);

    to_chars_result to_chars(char* first, char* last, float value, chars_format fmt);
    to_chars_result to_chars(char* first, char* last, double value, chars_format fmt);
    to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt);

    to_chars_result to_chars(char* first, char* last, float value, chars_format fmt, int precision);
    to_chars_result to_chars(char* first, char* last, double value, chars_format fmt, int precision);
    to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt, int precision);

    // 20.19.3, primitive numerical input conversion
    struct from_chars_result {
        const char* ptr;
        errc ec;
        friend bool operator==(const from_chars_result&, const from_chars_result&) = default;
    };

    from_chars_result from_chars(const char* first, const char* last, see below value, int base = 10);

    from_chars_result from_chars(const char* first, const char* last, float& value,
                                chars_format fmt = chars_format::general);
    from_chars_result from_chars(const char* first, const char* last, double& value,
                                chars_format fmt = chars_format::general);
    from_chars_result from_chars(const char* first, const char* last, long double& value,
                                chars_format fmt = chars_format::general);
}

1 The type `chars_format` is a bitmask type (16.3.3.3.4) with elements `scientific`, `fixed`, and `hex`.
2 The types `to_chars_result` and `from_chars_result` have the data members and special members specified above. They have no base classes or members other than those specified.

20.19.2 Primitive numeric output conversion

1 All functions named `to_chars` convert `value` into a character string by successively filling the range `[first, last)`, where `[first, last)` is required to be a valid range. If the member `ec` of the return value is such that the value is equal to the value of a value-initialized `errc`, the conversion was successful.
and the member \texttt{ptr} is the one-past-the-end pointer of the characters written. Otherwise, the member \texttt{ec} has the value \texttt{errc::value\_too\_large}, the member \texttt{ptr} has the value \texttt{last}, and the contents of the range \texttt{[first, last)} are unspecified.

The functions that take a floating-point \texttt{value} but not a \texttt{precision} parameter ensure that the string representation consists of the smallest number of characters such that there is at least one digit before the radix point (if present) and parsing the representation using the corresponding \texttt{from\_chars} function recovers \texttt{value} exactly.

\textit{[Note 1: This guarantee applies only if \texttt{to\_chars} and \texttt{from\_chars} are executed on the same implementation. — end note]}

If there are several such representations, the representation with the smallest difference from the floating-point argument value is chosen, resolving any remaining ties using rounding according to \texttt{round\_to\_nearest (17.3.4.1)}. 

The functions taking a \texttt{chars\_format} parameter determine the conversion specifier for \texttt{printf} as follows: The conversion specifier is \texttt{f} if \texttt{fmt} is \texttt{chars\_format::fixed}, \texttt{e} if \texttt{fmt} is \texttt{chars\_format::scientific}, \texttt{a} (without leading "0x" in the result) if \texttt{fmt} is \texttt{chars\_format::hex}, and \texttt{g} if \texttt{fmt} is \texttt{chars\_format::general}.

\begin{verbatim}
to_chars_result to_chars(char* first, char* last, see below value, int base = 10);
	extbf{Preconditions:} base has a value between 2 and 36 (inclusive).

\textbf{Effects:} The value of \texttt{value} is converted to a string of digits in the given base (with no redundant leading zeroes). Digits in the range 10..35 (inclusive) are represented as lowercase characters a..z. If \texttt{value} is less than zero, the representation starts with '‐'.

\textbf{Throws:} Nothing.

\textbf{Remarks:} The implementation shall provide overloads for all signed and unsigned integer types and \texttt{char} as the type of the parameter \texttt{value}.
\end{verbatim}

\begin{verbatim}
to_chars_result to_chars(char* first, char* last, float value);
to_chars_result to_chars(char* first, char* last, double value);
to_chars_result to_chars(char* first, char* last, long double value);

\textbf{Effects:} value is converted to a string in the style of \texttt{printf} in the "C" locale. The conversion specifier is \texttt{f} or \texttt{e}, chosen according to the requirement for a shortest representation (see above); a tie is resolved in favor of \texttt{f}.

\textbf{Throws:} Nothing.

\textbf{Remarks:} The implementation shall provide overloads for all signed and unsigned integer types and \texttt{char} as the type of the parameter \texttt{value}.
\end{verbatim}

\begin{verbatim}
to_chars_result to_chars(char* first, char* last, float value, chars_format fmt);
to_chars_result to_chars(char* first, char* last, double value, chars_format fmt);
to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt);

\textbf{Preconditions:} \texttt{fmt} has the value of one of the enumerators of \texttt{chars\_format}.

\textbf{Effects:} value is converted to a string in the style of \texttt{printf} in the "C" locale.

\textbf{Throws:} Nothing.

\textbf{Remarks:} The implementation shall provide overloads for all signed and unsigned integer types and \texttt{char} as the type of the parameter \texttt{value}.
\end{verbatim}

\begin{verbatim}
to_chars_result to_chars(char* first, char* last, float value, chars_format fmt, int precision);
to_chars_result to_chars(char* first, char* last, double value, chars_format fmt, int precision);
to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt, int precision);

\textbf{Preconditions:} \texttt{fmt} has the value of one of the enumerators of \texttt{chars\_format}.

\textbf{Effects:} value is converted to a string in the style of \texttt{printf} in the "C" locale with the given precision.

\textbf{Throws:} Nothing.

\textbf{Remarks:} The implementation shall provide overloads for all signed and unsigned integer types and \texttt{char} as the type of the parameter \texttt{value}.
\end{verbatim}

\textbf{20.19.3  Primitive numeric input conversion \texttt{[charconv.from.chars]} }

All functions named \texttt{from\_chars} analyze the string \texttt{[first, last)} for a pattern, where \texttt{[first, last)} is required to be a valid range. If no characters match the pattern, \texttt{value} is unmodified, the member \texttt{ptr} of the return value is \texttt{first} and the member \texttt{ec} is equal to \texttt{errc::invalid\_argument}.
[Note 1: If the pattern allows for an optional sign, but the string has no digit characters following the sign, no characters match the pattern. — end note]

Otherwise, the characters matching the pattern are interpreted as a representation of a value of the type of value. The member ptr of the return value points to the first character not matching the pattern, or has the value last if all characters match. If the parsed value is not in the range representable by the type of value, value is unmodified and the member ec of the return value is equal to errc::result_out_of_range. Otherwise, value is set to the parsed value, after rounding according to round_to_nearest (17.3.4.1), and the member ec is value-initialized.

```cpp
from_chars_result from_chars(const char* first, const char* last, value, int base = 10);
```

2 Preconditions: base has a value between 2 and 36 (inclusive).
3 Effects: The pattern is the expected form of the subject sequence in the "C" locale for the given nonzero base, as described for strtol, except that no "0x" or "0X" prefix shall appear if the value of base is 16, and except that '-' is the only sign that may appear, and only if value has a signed type.
4 Throws: Nothing.
5 Remarks: The implementation shall provide overloads for all signed and unsigned integer types and char as the referenced type of the parameter value.

```cpp
from_chars_result from_chars(const char* first, const char* last, float& value,
chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, double& value,
chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, long double& value,
chars_format fmt = chars_format::general);
```

6 Preconditions: fmt has the value of one of the enumerators of chars_format.
7 Effects: The pattern is the expected form of the subject sequence in the "C" locale, as described for strtod, except that
(7.1) the sign '+' may only appear in the exponent part;
(7.2) if fmt has chars_format::scientific set but not chars_format::fixed, the otherwise optional exponent part shall appear;
(7.3) if fmt has chars_format::fixed set but not chars_format::scientific, the optional exponent part shall not appear; and
(7.4) if fmt is chars_format::hex, the prefix "0x" or "0X" is assumed.

[Example 1: The string 0x123 is parsed to have the value 0 with remaining characters x123. — end example]

In any case, the resulting value is one of at most two floating-point values closest to the value of the string matching the pattern.

8 Throws: Nothing.

See also: ISO C 7.22.1.3, 7.22.1.4

20.20 Formatting

20.20.1 Header <format> synopsis

```cpp
namespace std {
  // 20.20.5.4, class template basic_format_context
template<class Out, class charT> class basic_format_context;
using format_context = basic_format_context<unspecialized, char>;
using wformat_context = basic_format_context<unspecialized, wchar_t>;

  // 20.20.6.3, class template basic_format_args
template<class Context> class basic_format_args;
using format_args = basic_format_args<format_context>;
using wformat_args = basic_format_args<wformat_context>;
```
template<class Out, class charT>
using format_args_t = basic_format_args<basic_format_context<Out, charT>>;

// 20.20.4, formatting functions

// 20.20.4.1, formatting functions

// 20.20.4.1.1, string_view

template<class... Args>
string format(string_view fmt, const Args&... args);

// 20.20.4.1.2, wstring_view

template<class... Args>
wstring format(wstring_view fmt, const Args&... args);

// 20.20.4.1.3, const locale&

template<class... Args>
string format(const locale& loc, string_view fmt, const Args&... args);

template<class... Args>
wstring format(const locale& loc, wstring_view fmt, const Args&... args);

// 20.20.4.1.4, string_view

string vformat(string_view fmt, format_args args);

// 20.20.4.1.5, wstring_view

wstring vformat(wstring_view fmt, wformat_args args);

// 20.20.4.1.6, const locale&

string vformat(const locale& loc, string_view fmt, format_args args);

wstring vformat(const locale& loc, wstring_view fmt, wformat_args args);

template<class Out, class... Args>
Out format_to(Out out, string_view fmt, const Args&... args);

template<class Out, class... Args>
Out format_to(Out out, wstring_view fmt, const Args&... args);

template<class Out, class... Args>
Out format_to(Out out, const locale& loc, string_view fmt, const Args&... args);

template<class Out, class... Args>
Out format_to(Out out, const locale& loc, wstring_view fmt, const Args&... args);

// 20.20.4.1.7, string_view

Out vformat_to(Out out, string_view fmt, format_args_t<type_identity_t<Out>, char> args);

Out vformat_to(Out out, string_view fmt, format_args_t<type_identity_t<Out>, wchar_t> args);

Out vformat_to(Out out, const locale& loc, string_view fmt, format_args_t<type_identity_t<Out>, char> args);

Out vformat_to(Out out, const locale& loc, wstring_view fmt, format_args_t<type_identity_t<Out>, wchar_t> args);

// 20.20.4.1.8, wstring_view

template<class Out> struct format_to_n_result {
Out out;
iter_difference_t<Out> size;
};

// 20.20.4.1.9, const locale&

template<class Out, class... Args>
format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n, string_view fmt, const Args&... args);

// 20.20.4.1.10, wstring_view

template<class Out, class... Args>
format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n, wstring_view fmt, const Args&... args);

// 20.20.4.1.11, const locale&

template<class Out, class... Args>
format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n, const locale& loc, string_view fmt, const Args&... args);

// 20.20.4.1.12, wstring_view

template<class Out, class... Args>
format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n, const locale& loc, wstring_view fmt, const Args&... args);

// 20.20.4.1.13, size_t

size_t formatted_size(string_view fmt, const Args&... args);

size_t formatted_size(wstring_view fmt, const Args&... args);
template<class... Args>
size_t formatted_size(const locale& loc, string_view fmt, const Args&... args);

// 20.20.5, formatter
template<class T, class charT = char> struct formatter;

// 20.20.5.3, class template basic_format_parse_context
template<class charT> class basic_format_parse_context;
using format_parse_context = basic_format_parse_context<char>;
using wformat_parse_context = basic_format_parse_context<wchar_t>;

// 20.20.6, arguments
// 20.20.6.1, class template basic_format_arg
template<class Context> class basic_format_arg;

template<class Visitor, class Context> see below visit_format_arg(Visitor& vis, basic_format_arg<Context> arg);

// 20.20.6.2, class template format-arg-store
template<class Context, class... Args> struct format-arg-store; // exposition only

template<class Context = format_context, class... Args>
format-arg-store<Context, Args...> make_format_arg(const Args&... args);

// 20.20.7, class format_error
class format_error;

1 The class template format_to_n_result has the template parameters, data members, and special members specified above. It has no base classes or members other than those specified.

20.20.2 Format string

A format string for arguments args is a (possibly empty) sequence of replacement fields, escape sequences, and characters other than { and }. Let charT be the character type of the format string. Each character that is not part of a replacement field or an escape sequence is copied unchanged to the output. An escape sequence is one of {, or }). It is replaced with { or }, respectively, in the output. The syntax of replacement fields is as follows:

replacement-field:
{ arg-id, opt format-specifier, opt }

arg-id:
0 positive-integer

positive-integer:
nonzero-digit positive-integer digit

nonnegative-integer:
digit
nonnegative-integer digit

nonzero-digit: one of
1 2 3 4 5 6 7 8 9

digit: one of
0 1 2 3 4 5 6 7 8 9

§ 20.20.2.1
format-specifier:
  : format-spec

format-spec:
  as specified by the formatter specialization for the argument type

2 The arg-id field specifies the index of the argument in args whose value is to be formatted and inserted into the output instead of the replacement field. If there is no argument with the index arg-id in args, the string is not a format string for args. The optional format-specifier field explicitly specifies a format for the replacement value.

3 [Example 1:
   string s = format("*{0}-*{1}", 8);  // value of s is "8-"
   — end example]

4 If all arg-ids in a format string are omitted (including those in the format-spec, as interpreted by the corresponding formatter specialization), argument indices 0, 1, 2, … will automatically be used in that order. If some arg-ids are omitted and some are present, the string is not a format string.

   [Note 1: A format string cannot contain a mixture of automatic and manual indexing. — end note]

   [Example 2:
   string s0 = format("{} to {}", "a", "b");  // OK, automatic indexing
   string s1 = format("{1} to {0}", "a", "b");  // OK, manual indexing
   string s2 = format("{0} to {}", "a", "b");  // not a format string (mixing automatic and manual indexing),
       // throws format_error
   string s3 = format("{} to {1}", "a", "b");  // not a format string (mixing automatic and manual indexing),
       // throws format_error
   — end example]

5 The format-spec field contains format specifications that define how the value should be presented. Each type can define its own interpretation of the format-spec field. If format-spec does not conform to the format specifications for the argument type referred to by arg-id, the string is not a format string for args.

   [Example 3:
   (5.1) — For arithmetic, pointer, and string types the format-spec is interpreted as a std-format-spec as described in (20.20.2.2).
   (5.2) — For chrono types the format-spec is interpreted as a chrono-format-spec as described in (27.12).
   (5.3) — For user-defined formatter specializations, the behavior of the parse member function determines how the format-spec is interpreted.
   — end example]

20.20.2.2 Standard format specifiers

1 Each formatter specializations described in 20.20.5.2 for fundamental and string types interprets format-spec as a std-format-spec.

   [Note 1: The format specification can be used to specify such details as field width, alignment, padding, and decimal precision. Some of the formatting options are only supported for arithmetic types. — end note]

The syntax of format specifications is as follows:

std-format-spec:
  fill-and-align_opt sign_opt #opt 0_opt width_opt precision_opt L_opt type_opt

fill-and-align:
  fill_opt align

fill:
  any character other than { or }

align: one of
  < > ^

sign: one of
  + - space

width:
  positive-integer
    { arg-id_opt }
precision:
  - nonnegative-integer
    - \{ arg-id_{opt} \}

type: one of
  a A b B c d e E f F g G o p s x X

2 [Note 2: The fill character can be any character other than \{ or \}. The presence of a fill character is signaled by the character following it, which must be one of the alignment options. If the second character of std-format-spec is not a valid alignment option, then it is assumed that both the fill character and the alignment option are absent. — end note]

3 The align specifier applies to all argument types. The meaning of the various alignment options is as specified in Table 59.

[Example 1:

```c
char c = 120;
string s0 = format("{:6}", 42); // value of s0 is " 42"
string s1 = format("{:6}"', 'x'); // value of s1 is "x"
string s2 = format("{:<6}"', 'x'); // value of s2 is "*****"
string s3 = format("{:^6}"', 'x'); // value of s3 is "*****x"
string s4 = format("{:*<6}"', 'x'); // value of s4 is "*****x"
string s5 = format("{:*>6}"', c); // value of s5 is "120"
string s6 = format("{:6}", true); // value of s6 is "true"
```
—end example]

[Note 3: Unless a minimum field width is defined, the field width is determined by the size of the content and the alignment option has no effect. — end note]

Table 59: Meaning of align options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Forces the field to be aligned to the start of the available space. This is the default for non-arithmetic types, charT, and bool, unless an integer presentation type is specified.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Forces the field to be aligned to the end of the available space. This is the default for arithmetic types other than charT and bool or when an integer presentation type is specified.</td>
</tr>
<tr>
<td>^</td>
<td>Forces the field to be centered within the available space by inserting ⌊ n \over 2 ⌋ characters before and ⌈ n \over 2 ⌉ characters after the value, where n is the total number of fill characters to insert.</td>
</tr>
</tbody>
</table>

4 The sign option is only valid for arithmetic types other than charT and bool or when an integer presentation type is specified. The meaning of the various options is as specified in Table 60.

Table 60: Meaning of sign options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Indicates that a sign should be used for both non-negative and negative numbers. The + sign is inserted before the output of to_chars for non-negative numbers other than negative zero. [Note 4: For negative numbers and negative zero the output of to_chars will already contain the sign so no additional transformation is performed. — end note]</td>
</tr>
<tr>
<td>-</td>
<td>Indicates that a sign should be used for negative numbers and negative zero only (this is the default behavior).</td>
</tr>
<tr>
<td>space</td>
<td>Indicates that a leading space should be used for non-negative numbers other than negative zero, and a minus sign for negative numbers and negative zero.</td>
</tr>
</tbody>
</table>

5 The sign option applies to floating-point infinity and NaN.

[Example 2:

```c
double inf = numeric_limits<
double>::infinity();
double nan = numeric_limits<
double>::quiet_NaN();
```

§ 20.20.2.2
string s0 = format("{0:},{0:+},{0:-},{0: }", 1); // value of s0 is "1,+1,1, 1"
string s1 = format("{0:},{0:+},{0:-},{0: }", -1); // value of s1 is "-1,-1,-1,-1"
string s2 = format("{0:},{0:+},{0:-},{0: }", inf); // value of s2 is "inf,+inf,inf, inf"
string s3 = format("{0:},{0:+},{0:-},{0: }", nan); // value of s3 is "nan,*nan,nan, nan"

— end example

6 The # option causes the alternate form to be used for the conversion. This option is valid for arithmetic types other than charT and bool or when an integer presentation type is specified, and not otherwise. For integral types, the alternate form inserts the base prefix (if any) specified in Table 62 into the output after the sign character (possibly space) if there is one, or before the output of to_chars otherwise. For floating-point types, the alternate form causes the result of the conversion of finite values to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for g and G conversions, trailing zeros are not removed from the result.

7 If { arg-id opt } is used in a width or precision, the value of the corresponding formatting argument is used in its place. If the corresponding formatting argument is not of integral type, or its value is negative for precision or non-positive for width, an exception of type format_error is thrown.

8 The positive-integer in width is a decimal integer defining the minimum field width. If width is not specified, there is no minimum field width, and the field width is determined based on the content of the field.

9 The width of a string is defined as the estimated number of column positions appropriate for displaying it in a terminal.

[Note 5: This is similar to the semantics of the POSIX wcswidth function. — end note]

10 For the purposes of width computation, a string is assumed to be in a locale-independent, implementation-defined encoding. Implementations should use a Unicode encoding on platforms capable of displaying Unicode text in a terminal.

[Note 6: This is the case for Windows226-based and many POSIX-based operating systems. — end note]

11 For a string in a Unicode encoding, implementations should estimate the width of a string as the sum of estimated widths of the first code points in its extended grapheme clusters. The extended grapheme clusters of a string are defined by UAX #29. The estimated width of the following code points is 2:

(11.1) — U+1100-U+115F
(11.2) — U+2329-U+232A
(11.3) — U+2E80-U+303E
(11.4) — U+3040-U+A4CF
(11.5) — U+AC00-U+D7A3
(11.6) — U+FE10-U+FE19
(11.7) — U+1F300-U+1F64F
(11.8) — U+1F900-U+1F9FF
(11.9) — U+20000-U+2FFFD
(11.10) — U+30000-U+3FFFD

The estimated width of other code points is 1.

12 For a string in a non-Unicode encoding, the width of a string is unspecified.

13 A zero (0) character preceding the width field pads the field with leading zeros (following any indication of sign or base) to the field width, except when applied to an infinity or NaN. This option is only valid

226) Windows® is a registered trademark of Microsoft Corporation. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.
for arithmetic types other than charT and bool or when an integer presentation type is specified. If the 0 character and an align option both appear, the 0 character is ignored.

[Example 3:

```cpp
class char c = 120;
string s1 = format("{:+06d}", c); // value of s1 is "+00120"
string s2 = format("{:#06x}", 0xa); // value of s2 is "0x000a"
string s3 = format("{:<06}", -42); // value of s3 is "-42 " (0 is ignored because of < alignment)
```
— end example]

14 The nonnegative-integer in precision is a decimal integer defining the precision or maximum field size. It can only be used with floating-point and string types. For floating-point types this field specifies the formatting precision. For string types, this field provides an upper bound for the estimated width of the prefix of the input string that is copied into the output. For a string in a Unicode encoding, the formatter copies to the output the longest prefix of whole extended grapheme clusters whose estimated width is no greater than the precision.

15 When the L option is used, the form used for the conversion is called the locale-specific form. The L option is only valid for arithmetic types, and its effect depends upon the type.

(15.1) — For integral types, the locale-specific form causes the context’s locale to be used to insert the appropriate digit group separator characters.

(15.2) — For floating-point types, the locale-specific form causes the context’s locale to be used to insert the appropriate digit group and radix separator characters.

(15.3) — For the textual representation of bool, the locale-specific form causes the context’s locale to be used to insert the appropriate string as if obtained with numpunct::truename or numpunct::falsename.

16 The type determines how the data should be presented.

17 The available string presentation types are specified in Table 61.

Table 61: Meaning of type options for strings

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none, s</td>
<td>Copies the string to the output.</td>
</tr>
</tbody>
</table>

18 The meaning of some non-string presentation types is defined in terms of a call to to_chars. In such cases, let [first, last) be a range large enough to hold the to_chars output and value be the formatting argument value. Formatting is done as if by calling to_chars as specified and copying the output through the output iterator of the format context.

[Note 7: Additional padding and adjustments are performed prior to copying the output through the output iterator as specified by the format specifiers. — end note]

19 The available integer presentation types for integral types other than bool and charT are specified in Table 62.

[Example 4:

```cpp
string s0 = format("{}", 42); // value of s0 is "42"
string s1 = format("{:b} {:d} {:o} {:x}", 42); // value of s1 is "101010 42 52 2a"
string s2 = format("{:#x} {:#X}", 42); // value of s2 is "0x2a 0X2A"
string s3 = format("{:L}", 1234); // value of s3 might be "1,234" (depending on the locale)
```
— end example]

20 The available charT presentation types are specified in Table 63.

21 The available bool presentation types are specified in Table 64.

22 The available floating-point presentation types and their meanings for values other than infinity and NaN are specified in Table 65. For lower-case presentation types, infinity and NaN are formatted as inf and nan, respectively. For upper-case presentation types, infinity and NaN are formatted as INF and NAN, respectively.

[Note 9: In either case, a sign is included if indicated by the sign option. — end note]

23 The available pointer presentation types and their mapping to to_chars are specified in Table 66.

[Note 10: Pointer presentation types also apply to nullptr_t. — end note]
Table 62: Meaning of type options for integer types [tab:format.type.int]

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>to_chars(first, last, value, 2); the base prefix is 0b.</td>
</tr>
<tr>
<td>B</td>
<td>The same as b, except that the base prefix is 0B.</td>
</tr>
<tr>
<td>c</td>
<td>Copies the character static_cast&lt;charT&gt;(value) to the output. Throws format_error if value is not in the range of representable values for charT.</td>
</tr>
<tr>
<td>d</td>
<td>to_chars(first, last, value).</td>
</tr>
<tr>
<td>o</td>
<td>to_chars(first, last, value, 8); the base prefix is 0 if value is nonzero and is empty otherwise.</td>
</tr>
<tr>
<td>x</td>
<td>to_chars(first, last, value, 16); the base prefix is 0x.</td>
</tr>
<tr>
<td>X</td>
<td>The same as x, except that it uses uppercase letters for digits above 9 and the base prefix is 0X.</td>
</tr>
<tr>
<td>none</td>
<td>The same as d.</td>
</tr>
</tbody>
</table>

[Note 8: If the formatting argument type is charT or bool, the default is instead c or s, respectively. — end note]

Table 63: Meaning of type options for charT [tab:format.type.char]

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none, c</td>
<td>Copies the character to the output.</td>
</tr>
<tr>
<td>b, B, d, o, x, X</td>
<td>As specified in Table 62.</td>
</tr>
</tbody>
</table>

Table 64: Meaning of type options for bool [tab:format.type.bool]

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none, s</td>
<td>Copies textual representation, either true or false, to the output.</td>
</tr>
<tr>
<td>b, B, c, d, o, x, X</td>
<td>As specified in Table 62 for the value static_cast&lt;unsigned char&gt;(value).</td>
</tr>
</tbody>
</table>

Table 65: Meaning of type options for floating-point types [tab:format.type.float]

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>If precision is specified, equivalent to to_chars(first, last, value, chars_format::hex, precision) where precision is the specified formatting precision; equivalent to to_chars(first, last, value, chars_format::hex) otherwise.</td>
</tr>
<tr>
<td>A</td>
<td>The same as a, except that it uses uppercase letters for digits above 9 and P to indicate the exponent.</td>
</tr>
<tr>
<td>e</td>
<td>Equivalent to to_chars(first, last, value, chars_format::scientific, precision) where precision is the specified formatting precision, or 6 if precision is not specified.</td>
</tr>
<tr>
<td>E</td>
<td>The same as e, except that it uses E to indicate exponent.</td>
</tr>
<tr>
<td>f, F</td>
<td>Equivalent to to_chars(first, last, value, chars_format::fixed, precision) where precision is the specified formatting precision, or 6 if precision is not specified.</td>
</tr>
<tr>
<td>g</td>
<td>Equivalent to to_chars(first, last, value, chars_format::general, precision) where precision is the specified formatting precision, or 6 if precision is not specified.</td>
</tr>
<tr>
<td>G</td>
<td>The same as g, except that it uses E to indicate exponent.</td>
</tr>
<tr>
<td>none</td>
<td>If precision is specified, equivalent to to_chars(first, last, value, chars_format::general, precision) where precision is the specified formatting precision; equivalent to to_chars(first, last, value) otherwise.</td>
</tr>
</tbody>
</table>
Table 66: Meaning of type options for pointer types  [tab:format.type.ptr]

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none, p</td>
<td>If uintptr_t is defined, to_chars(first, last, reinterpret_cast&lt;uintptr_t&gt;(value), 16) with the prefix 0x added to the output; otherwise, implementation-defined.</td>
</tr>
</tbody>
</table>

20.20.3 Error reporting  [format.err.report]

Formatting functions throw format_error if an argument fmt is passed that is not a format string for args. They propagate exceptions thrown by operations of formatter specializations and iterators. Failure to allocate storage is reported by throwing an exception as described in 16.4.6.13.

20.20.4 Formatting functions  [format.functions]

In the description of the functions, operator + is used for some of the iterator categories for which it does not have to be defined. In these cases the semantics of a + n are the same as in 25.2.

```
template<class... Args>
string format(string_view fmt, const Args&... args);
Effects: Equivalent to:
return vformat(fmt, make_format_args(args...));
```

```
template<class... Args>
wstring format(wstring_view fmt, const Args&... args);
Effects: Equivalent to:
return vformat(fmt, make_wformat_args(args...));
```

```
template<class... Args>
string format(const locale& loc, string_view fmt, const Args&... args);
Effects: Equivalent to:
return vformat(loc, fmt, make_format_args(args...));
```

```
template<class... Args>
wstring format(const locale& loc, wstring_view fmt, const Args&... args);
Effects: Equivalent to:
return vformat(loc, fmt, make_wformat_args(args...));
```

```
string vformat(string_view fmt, format_args args);
wstring vformat(wstring_view fmt, wformat_args args);
string vformat(const locale& loc, string_view fmt, format_args args);
wstring vformat(const locale& loc, wstring_view fmt, wformat_args args);
Returns: A string object holding the character representation of formatting arguments provided by args formatted according to specifications given in fmt. If present, loc is used for locale-specific formatting.
Throws: As specified in 20.20.3.
```

```
template<class Out, class... Args>
Out format_to(Out out, string_view fmt, const Args&... args);
template<class Out, class... Args>
Out format_to(Out out, wstring_view fmt, const Args&... args);
Effects: Equivalent to:
using context = basic_format_context<Out, decltype(fmt)::value_type>;
return vformat_to(out, fmt, make_format_args<context>(args...));
```

```
template<class Out, class... Args>
Out format_to(Out out, const locale& loc, string_view fmt, const Args&... args);
template<class Out, class... Args>
Out format_to(Out out, const locale& loc, wstring_view fmt, const Args&... args);
Effects: Equivalent to:
```
using context = basic_format_context<Out, decltype(fmt)::value_type>;
return vformat_to(out, loc, fmt, make_format_args<context>(args...));

template<class Out>
Out vformat_to(Out out, string_view fmt,
        format_args_t<type_identity_t<Out>, char> args);

template<class Out>
Out vformat_to(Out out, wstring_view fmt,
        format_args_t<type_identity_t<Out>, wchar_t> args);

template<class Out>
Out vformat_to(Out out, const locale& loc, string_view fmt,
        format_args_t<type_identity_t<Out>, char> args);

template<class Out>
Out vformat_to(Out out, const locale& loc, wstring_view fmt,
        format_args_t<type_identity_t<Out>, wchar_t> args);

Let charT be decltype(fmt)::value_type.

Constraints: Out satisfies output_iterator<const charT&>.

Preconditions: Out models output_iterator<const charT&>.

Effects: Places the character representation of formatting the arguments provided by args, formatted according to the specifications given in fmt, into the range [out, out + N), where N is formatted_size(fmt, args...) for the functions without a loc parameter and formatted_size(loc, fmt, args...) for the functions with a loc parameter. If present, loc is used for locale-specific formatting.

Returns: out + N.

Throws: As specified in 20.20.3.

template<class Out, class... Args>
format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
        string_view fmt, const Args&... args);

template<class Out, class... Args>
format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
        wstring_view fmt, const Args&... args);

template<class Out, class... Args>
format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
        const locale& loc, string_view fmt,
        const Args&... args);

template<class Out, class... Args>
format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
        const locale& loc, wstring_view fmt,
        const Args&... args);

Let

(16.1) charT be decltype(fmt)::value_type,

(16.2) N be formatted_size(fmt, args...) for the functions without a loc parameter and formatted_size(loc, fmt, args...) for the functions with a loc parameter, and

(16.3) M be clamp(n, 0, N).

Constraints: Out satisfies output_iterator<const charT&>.

Preconditions: Out models output_iterator<const charT&>, and formatter<Ti, charT> meets the Formatter requirements (20.20.5.1) for each Ti in Args.

Effects: Places the first M characters of the character representation of formatting the arguments provided by args, formatted according to the specifications given in fmt, into the range [out, out + M). If present, loc is used for locale-specific formatting.

Returns: {out + M, N}.

Throws: As specified in 20.20.3.

template<class... Args>
size_t formatted_size(string_view fmt, const Args&... args);

template<class... Args>
size_t formatted_size(wstring_view fmt, const Args&... args);
template<class... Args>
size_t formatted_size(const locale& loc, string_view fmt, const Args&... args);

template<class... Args>
size_t formatted_size(const locale& loc, wstring_view fmt, const Args&... args);

Let charT be decltype(fmt)::value_type.

Preconditions: formatter<Ti, charT> meets the Formatter requirements (20.20.5.1) for each Ti in Args.

Returns: The number of characters in the character representation of formatting arguments args formatted according to specifications given in fmt. If present, loc is used for locale-specific formatting.

Throws: As specified in 20.20.3.

20.20.5 Formatter

20.20.5.1 Formatter requirements

A type F meets the Formatter requirements if:

1. it meets the
   (1.1) — Cpp17DefaultConstructible (Table 27),
   (1.1.1) — Cpp17CopyConstructible (Table 29),
   (1.1.2) — Cpp17CopyAssignable (Table 31), and
   (1.1.3) — Cpp17Destructible (Table 32) requirements,
(1.2) — it is swappable (16.4.4.3) for lvalues, and
(1.3) — the expressions shown in Table 67 are valid and have the indicated semantics.

2. Given character type charT, output iterator type Out, and formatting argument type T, in Table 67:
   (2.1) — f is a value of type F,
   (2.2) — u is an lvalue of type T,
   (2.3) — t is a value of a type convertible to (possibly const) T,
   (2.4) — PC is basic_format_parse_context<charT>,
   (2.5) — FC is basic_format_context<Out, charT>,
   (2.6) — pc is an lvalue of type PC, and
   (2.7) — fc is an lvalue of type FC.

pc.begin() points to the beginning of the format-spec (20.20.2) of the replacement field being formatted in the format string. If format-spec is empty then either pc.begin() == pc.end() or *pc.begin() == '}'.

20.20.5.2 Formatter specializations

The functions defined in 20.20.4 use specializations of the class template formatter to format individual arguments.

Let charT be either char or wchar_t. Each specialization of formatter is either enabled or disabled, as described below.

[Note 1: Enabled specializations meet the Formatter requirements, and disabled specializations do not. — end note]

Each header that declares the template formatter provides the following enabled specializations:

1. The specializations
   template<> struct formatter<char, char>;
   template<> struct formatter<char, wchar_t>;
   template<> struct formatter<wchar_t, wchar_t>;

2. For each charT, the string type specializations
   template<> struct formatter<charT*, charT>;
   template<> struct formatter<const charT*, charT>;
   template<size_t N> struct formatter<const charT[N], charT>;

§ 20.20.5.2
### Table 67: Formatter requirements  

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
</table>
| `f.parse(pc)`  | PC::iterator| Parses *format-spec* (20.20.2) for type `T` in the range `[pc.begin(), pc.end())` until the first unmatched character. Throws `format_error` unless the whole range is parsed or the unmatched character is `{`.  

*Note 1:* This allows formatters to emit meaningful error messages. — end note  
Stores the parsed format specifiers in `*this` and returns an iterator past the end of the parsed range. |
| `f.format(t, fc)` | FC::iterator | Formats `t` according to the specifiers stored in `*this`, writes the output to `fc.out()` and returns an iterator past the end of the output range. The output shall only depend on `t`, `fc.locale()`, and the range `[pc.begin(), pc.end())` from the last call to `f.parse(pc)`. |
| `f.format(u, fc)` | FC::iterator | As above, but does not modify `u`. |

**template<class traits, class Allocator>**  
`struct formatter<basic_string<charT, traits, Allocator>, charT>;`  
`template<class traits>  
struct formatter<basic_string_view<charT, traits>, charT>;`  

(2.3) For each `charT`, for each cv-unqualified arithmetic type `ArithmeticT` other than `char`, `wchar_t`, `char8_t`, `char16_t`, or `char32_t`, a specialization  
`template<> struct formatter<ArithmeticT, charT>;`  

(2.4) For each `charT`, the pointer type specializations  
`template<> struct formatter<nullptr_t, charT>;`  
`template<> struct formatter<void*, charT>;`  
`template<> struct formatter<const void*, charT>;`  

The `parse` member functions of these formatters interpret the format specification as a *std-format-spec* as described in 20.20.2.2.  

*Note 2:* Specializations such as `formatter<wchar_t, char>` and `formatter<const char*, wchar_t>` that would require implicit multibyte / wide string or character conversion are disabled. — end note  

3 For any types `T` and `charT` for which neither the library nor the user provides an explicit or partial specialization of the class template `formatter`, `formatter<T, charT>` is disabled.  

4 If the library provides an explicit or partial specialization of `formatter<T, charT>`, that specialization is enabled except as noted otherwise.  

5 If `F` is a disabled specialization of `formatter`, these values are `false`:  

(5.1) `is_default_constructible_v<F>`,  
(5.2) `is_copy_constructible_v<F>`,  
(5.3) `is_move_constructible_v<F>`,  
(5.4) `is_copyAssignable_v<F>`, and  
(5.5) `is_move_assignable_v<F>`.  

6 An enabled specialization `formatter<T, charT>` meets the *Formatter* requirements (20.20.5.1).  

**Example 1:**  
```cpp  
#include <format>  
enum color { red, green, blue };  
const char* color_names[] = { "red", "green", "blue" };  
```
template<>
struct std::formatter<color> : std::formatter<const char*> {
    auto format(color c, format_context& ctx) {
        return formatter<const char*>::format(color_names[c], ctx);
    }
};

struct err {};

std::string s0 = std::format("{}", 42);    // OK, library-provided formatter
std::string s1 = std::format("{}", L"foo"); // error: disabled formatter
std::string s2 = std::format("{}", red);    // OK, user-provided formatter
std::string s3 = std::format("{}", err());  // error: disabled formatter

— end example

20.20.5.3 Class template basic_format_parse_context

namespace std {
    template<class charT>
    class basic_format_parse_context {
    public:
        using char_type = charT;
        using const_iterator = typename basic_string_view<charT>::const_iterator;
        using iterator = const_iterator;

        private:
            iterator begin_;    // exposition only
            iterator end_;      // exposition only
            enum indexing { unknown, manual, automatic };    // exposition only
            indexing indexing_; // exposition only
            size_t next_arg_id_;  // exposition only
            size_t num_args_;    // exposition only

        public:
            constexpr explicit basic_format_parse_context(basic_string_view<charT> fmt, size_t num_args = 0) noexcept;
            basic_format_parse_context(const basic_format_parse_context&) = delete;
            basic_format_parse_context& operator=(const basic_format_parse_context&) = delete;

            constexpr const_iterator begin() const noexcept;
            constexpr const_iterator end() const noexcept;
            constexpr void advance_to(const_iterator it);
            constexpr size_t next_arg_id();
            constexpr void check_arg_id(size_t id);
        }
    } // namespace std

1 An instance of basic_format_parse_context holds the format string parsing state consisting of the format string range being parsed and the argument counter for automatic indexing.

constexpr explicit basic_format_parse_context(basic_string_view<charT> fmt,
                                               size_t num_args = 0) noexcept;

2 Effects: Initializes begin_ with fmt.begin(), end_ with fmt.end(), indexing_ with unknown, next_arg_id_ with 0, and num_args_ with num_args.

constexpr const_iterator begin() const noexcept;

3 Returns: begin_.

constexpr const_iterator end() const noexcept;

4 Returns: end_.

constexpr void advance_to(const_iterator it);

5 Preconditions: end() is reachable from it.
Effects: Equivalent to: `begin_ = it;` 
`constexpr size_t next_arg_id();`

Effects: If `indexing_ != manual`, equivalent to:
```
    if (indexing_ == unknown)
        indexing_ = automatic;
    return next_arg_id_++;
```

Throws: `format_error` if `indexing_ == manual` which indicates mixing of automatic and manual argument indexing.

`constexpr void check_arg_id(size_t id);`

Effects: If `indexing_ != automatic`, equivalent to:
```
    if (indexing_ == unknown)
        indexing_ = manual;
```

Throws: `format_error` if `indexing_ == automatic` which indicates mixing of automatic and manual argument indexing.

Remarks: Call expressions where `id >= num_args_` are not core constant expressions (7.7).

20.20.5.4 Class template `basic_format_context` 

```cpp
namespace std {
    template<class Out, class charT>
    class basic_format_context {
        basic_format_args<basic_format_context> args_;    // exposition only
        Out out_;                                         // exposition only

    public:
        using iterator = Out;
        using char_type = charT;
        template<class T> using formatter_type = formatter<T, charT>;

        basic_format_arg<basic_format_context> arg(size_t id) const;
        std::locale locale();
        iterator out();
        void advance_to(iterator it);
    }
}
```

An instance of `basic_format_context` holds formatting state consisting of the formatting arguments and the output iterator.

`Out` shall model `output_iterator<const charT&>`.

`format_context` is an alias for a specialization of `basic_format_context` with an output iterator that appends to `string`, such as `back_insert_iterator<string>`. Similarly, `wformat_context` is an alias for a specialization of `basic_format_context` with an output iterator that appends to `wstring`. 

[Note 1: For a given type `charT`, implementations are encouraged to provide a single instantiation of `basic_format_context` for appending to `basic_string<charT>`, `vector<charT>`, or any other container with contiguous storage by wrapping those in temporary objects with a uniform interface (such as a `span<charT>`) and polymorphic reallocation. —end note]

`basic_format_arg<basic_format_context> arg(size_t id) const;`

Returns: `args_.get(id)`. 

`std::locale locale();`

Returns: The locale passed to the formatting function if the latter takes one, and `std::locale()` otherwise.

`iterator out();`

Returns: `out_`. 

§ 20.20.5.4
void advance_to(iterator it);

Effects: Equivalent to: out_ = it;

[Example 1:

struct S { int value; };

template<> struct std::formatter<S> {
    size_t width_arg_id = 0;

    // Parses a width argument id in the format { digit }.
    constexpr auto parse(format_parse_context& ctx) {
        auto iter = ctx.begin();
        auto get_char = [&](){ return iter != ctx.end() ? *iter : 0; }; 
        if (get_char() != '{') 
            return iter;
        ++iter;
        char c = get_char();
        if (!isdigit(c) || (++iter, get_char()) != '}')
            throw format_error("invalid format");
        width_arg_id = c - '0';
        ctx.check_arg_id(width_arg_id);
        return ++iter;
    }

    // Formats an S with width given by the argument width_arg_id.
    auto format(S s, format_context& ctx) {
        int width = visit_format_arg([](auto value) -> int {
            if constexpr (!is_integral_v<decltype(value)>)
                throw format_error("width is not integral");
            else if (value < 0 || value > numeric_limits<int>::max())
                throw format_error("invalid width");
            else
                return value;
        }, ctx.arg(width_arg_id));
        return format_to(ctx.out(), "{0:x<{1}}", s.value, width);
    }
};

std::string s = std::format("{0:{1}}", S{42}, 10); // value of s is "xxxxxxxx42"

— end example]
template<class traits>
  explicit basic_format_arg(
    basic_string_view<char_type, traits> s) noexcept;  // exposition only

template<class traits, class Allocator>
  explicit basic_format_arg(
    const basic_string<char_type, traits, Allocator>& s) noexcept;  // exposition only
explicit basic_format_arg(nullptr_t) noexcept;  // exposition only

template<class T>
  explicit basic_format_arg(const T* p) noexcept;  // exposition only

public:
  basic_format_arg() noexcept;
  explicit operator bool() const noexcept;
};

An instance of basic_format_arg provides access to a formatting argument for user-defined formatters.

The behavior of a program that adds specializations of basic_format_arg is undefined.

Postconditions: !(this).

template<class T> explicit basic_format_arg(const T& v) noexcept;

Constraints: The template specialization
typename Context::template formatter_type<T>
meets the Formatter requirements (20.20.5.1). The extent to which an implementation determines
that the specialization meets the Formatter requirements is unspecified, except that as a minimum the expression
typename Context::template formatter_type<T>()
.format(declval<const T&>(), declval<Context&>())
shall be well-formed when treated as an unevaluated operand.

Effects:

(5.1) if T is bool or char_type, initializes value with v;
(5.2) otherwise, if T is char and char_type is wchar_t, initializes value with static_cast<wchar_t>(v);
(5.3) otherwise, if T is a signed integer type (6.8.2) and sizeof(T) <= sizeof(int), initializes value with static_cast<int>(v);
(5.4) otherwise, if T is an unsigned integer type and sizeof(T) <= sizeof(unsigned int), initializes value with static_cast<unsigned int>(v);
(5.5) otherwise, if T is a signed integer type and sizeof(T) <= sizeof(long long int), initializes value with static_cast<long long int>(v);
(5.6) otherwise, if T is an unsigned integer type and sizeof(T) <= sizeof(unsigned long long int), initializes value with static_cast<unsigned long long int>(v);
(5.7) otherwise, initializes value with handle(v).

explicit basic_format_arg(float n) noexcept;
explicit basic_format_arg(double n) noexcept;
explicit basic_format_arg(long double n) noexcept;

Effects: Initializes value with n.

explicit basic_format_arg(const char_type* s);

Preconditions: s points to a NTCTS (3.36).
Effects: Initializes value with \( s \).

```cpp
template<class traits>
explicit basic_format_arg(basic_string_view<char_type, traits> s) noexcept;
```

Effects: Initializes value with \( s \).  

```cpp
template<class traits, class Allocator>
explicit basic_format_arg(
    const basic_string<char_type, traits, Allocator>& s) noexcept;
```

Effects: Initializes value with \( \text{basic_string_view}(s.\text{data}(), s.\text{size}) \).

```cpp
explicit basic_format_arg(nullptr_t) noexcept;
```

Effects: Initializes value with \( \text{static_cast<const void*>(nullptr)} \).

```cpp
template<class T> explicit basic_format_arg(const T* p) noexcept;
```

Constraints: \( \text{is_void_v<T>} \) is true.

Effects: Initializes value with \( p \).

[Note 1: Constructing \( \text{basic_format_arg} \) from a pointer to a member is ill-formed unless the user provides an enabled specialization of \( \text{formatter} \) for that pointer to member type. — end note]

Explicit operator bool() const noexcept;

Returns: \( !\text{holds_alternative<monostate>}(value) \).

The class `handle` allows formatting an object of a user-defined type.

```cpp
namespace std {
    template<class Context>
    class basic_format_arg<Context>::handle {
    const void* ptr_; // exposition only
    void (*format_)(basic_format_parse_context<char_type>&,
                    Context&, const void*); // exposition only
    template<class T> explicit handle(const T& val) noexcept; // exposition only
    friend class basic_format_arg<Context>; // exposition only
    public:
        void format(basic_format_parse_context<char_type>&, Context& ctx) const;
    }
}
```

Effects: Initializes \( \text{ptr} \) with \( \text{addressof} \) and \( \text{format} \) with

```cpp
[](basic_format_parse_context<char_type>& parse_ctx,
    Context& format_ctx, const void* ptr) {
    typename Context::template formatter_type<T> f;
    parse_ctx.advance_to(f.parse(parse_ctx));
    format_ctx.advance_to(f.format(*static_cast<const T*>(ptr), format_ctx));
}
```

Effects: Equivalent to: \( \text{format}_\text{ctx}(\text{parse}_\text{ctx}, \text{format}_\text{ctx}, \text{ptr}) \);

```cpp
template<class Visitor, class Context>
see below visit_format_arg(Visitor& vis, basic_format_arg<Context> arg);
```

Effects: Equivalent to: \( \text{return visit(forward<Visitor>(vis), arg.value)} \);

### 20.20.6.2 Class template format-arg-store [format.arg.store]

```cpp
namespace std {
    template<class Context, class... Args>
    struct format-arg-store { // exposition only
```
An instance of `format-arg-store` stores formatting arguments.

```cpp
template<class Context = format_context, class... Args>
format-arg-store<Context, Args...> make_format_args(const Args&... args);
```

**Preconditions:** The type `typename Context::template formatter_type<T_i>` meets the `Formatter` requirements (20.20.5.1) for each `T_i` in `Args`.

**Returns:** `{basic_format_arg<Context>(args)...}`.

**Effects:** Equivalent to: return `make_format_args<wformat_context>(args...);`

### 20.20.6.3 Class template basic_format_args

```cpp
namespace std {
    template<class Context>
    class basic_format_args {
        size_t size_;                   // exposition only
        const basic_format_arg<Context>* data_;     // exposition only

        public:
            basic_format_args() noexcept;
            template<class... Args>
            basic_format_args(const format-arg-store<Context, Args...>& store) noexcept;

            basic_format_arg<Context> get(size_t i) const noexcept;
    };
}
```

An instance of `basic_format_args` provides access to formatting arguments.

**`basic_format_args()` noexcept;**

**Effects:** Initializes `size_` with 0.

**template<class... Args>**

```cpp
basic_format_args(const format-arg-store<Context, Args...>& store) noexcept;
```

**Effects:** Initializes `size_` with `sizeof...(Args)` and `data_` with `store.args.data()`.

**basic_format_arg<Context> get(size_t i) const noexcept;**

**Returns:** `i < size_ ? data_[i] : basic_format_arg<Context>()`.

[Note 1: Implementations are encouraged to optimize the representation of `basic_format_args` for small number of formatting arguments by storing indices of type alternatives separately from values and packing the former. — end note]

### 20.20.7 Class format_error

```cpp
namespace std {
    class format_error : public runtime_error {
    public:
        explicit format_error(const string& what_arg);
        explicit format_error(const char* what_arg);
    };
}
```

The class `format_error` defines the type of objects thrown as exceptions to report errors from the formatting library.

**`format_error(const string& what_arg);`**

**Postconditions:** `strcmp(what(), what_arg.c_str()) == 0.`
format_error(const char* what_arg);

3  Postconditions: strcmp(what(), what_arg) == 0.
21 Strings library

21.1 General

This Clause describes components for manipulating sequences of any non-array trivial standard-layout (6.8) type. Such types are called char-like types, and objects of char-like types are called char-like objects or simply characters.

The following subclauses describe a character traits class, string classes, and null-terminated sequence utilities, as summarized in Table 68.

Table 68: Strings library summary

<table>
<thead>
<tr>
<th>Subclause Header</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.2 Character traits</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>21.3 String classes</td>
<td>&lt;string_view&gt;</td>
</tr>
<tr>
<td>21.4 String view classes</td>
<td>&lt;cctype&gt;, &lt;cstdlib&gt;, &lt;cstring&gt;, &lt;cuchar&gt;, &lt;cwchar&gt;, &lt;cwctype&gt;</td>
</tr>
<tr>
<td>21.5 Null-terminated sequence utilities</td>
<td></td>
</tr>
</tbody>
</table>

21.2 Character traits

21.2.1 General

Subclause 21.2 defines requirements on classes representing character traits, and defines a class template char_traits<charT>, along with five specializations, char_traits<char>, char_traits<char8_t>, char_traits<char16_t>, char_traits<char32_t>, and char_traits<wchar_t>, that meet those requirements.

Most classes specified in 21.3, 21.4, and Clause 29 need a set of related types and functions to complete the definition of their semantics. These types and functions are provided as a set of member typedef-names and functions in the template parameter traits used by each such template. Subclause 21.2 defines the semantics of these members.

To specialize those templates to generate a string, string view, or iostream class to handle a particular character container type (3.10) C, that and its related character traits class X are passed as a pair of parameters to the string, string view, or iostream template as parameters charT and traits. If X::char_type is not the same type as C, the program is ill-formed.

21.2.2 Character traits requirements

In Table 69, X denotes a traits class defining types and functions for the character container type C; c and d denote values of type C; p and q denote values of type const C*; s denotes a value of type C*; n, i and j denote values of type size_t; e and f denote values of type X::int_type; pos denotes a value of type X::pos_type; and r denotes an lvalue of type C. Operations on X shall not throw exceptions.

Table 69: Character traits requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::char_type</td>
<td>C</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::int_type</td>
<td>(described in 21.2.3)</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::off_type</td>
<td>(described in 29.2.2 and 29.3)</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::pos_type</td>
<td>(described in 29.2.2 and 29.3)</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::state_type</td>
<td>(described in 21.2.3)</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::eq(c,d)</td>
<td>bool</td>
<td>Returns: whether c is to be treated as equal to d.</td>
<td>constant</td>
</tr>
<tr>
<td>X::lt(c,d)</td>
<td>bool</td>
<td>Returns: whether c is to be treated as less than d.</td>
<td>constant</td>
</tr>
</tbody>
</table>
Table 69: Character traits requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::compare(p,q,n)</td>
<td>int</td>
<td>Returns: 0 if for each i in [0,n), X::eq(p[i],q[i]) is true; else, a negative value if, for some j in [0,n), X::lt(p[j],q[j]) is true and for each i in [0,j) X::eq(p[i],q[i]) is true; else a positive value.</td>
<td>linear</td>
</tr>
<tr>
<td>X::length(p)</td>
<td>size_t</td>
<td>Returns: the smallest i such that X::eq(p[i],charT()) is true.</td>
<td>linear</td>
</tr>
<tr>
<td>X::find(p,n,c)</td>
<td>const X::char_type*</td>
<td>Returns: the smallest q in [p,p+n) such that X::eq(*q,c) is true, zero otherwise.</td>
<td>linear</td>
</tr>
<tr>
<td>X::move(s,p,n)</td>
<td>X::char_type*</td>
<td>for each i in [0,n), performs X::assign(s[i],p[i]). Copies correctly even where the ranges [p,p+n) and [s,s+n) overlap. Returns: s.</td>
<td>linear</td>
</tr>
<tr>
<td>X::copy(s,p,n)</td>
<td>X::char_type*</td>
<td>Preconditions: p not in [s,s+n). Returns: s. for each i in [0,n), performs X::assign(s[i],p[i]).</td>
<td>linear</td>
</tr>
<tr>
<td>X::assign(r,d)</td>
<td>(not used)</td>
<td>assigns r=d.</td>
<td>constant</td>
</tr>
<tr>
<td>X::assign(s,n,c)</td>
<td>X::char_type*</td>
<td>for each i in [0,n), performs X::assign(s[i],c). Returns: s.</td>
<td>linear</td>
</tr>
<tr>
<td>X::not_eof(e)</td>
<td>int_type</td>
<td>Returns: e if X::eq_int_type(e,X::eof()) is false, otherwise a value f such that X::eq_int_type(f,X::eof()) is false.</td>
<td>constant</td>
</tr>
<tr>
<td>X::to_char_type(e)</td>
<td>X::char_type</td>
<td>Returns: if for some c, X::eq_int_type(e,X::to_int_type(c)) is true; c else some unspecified value.</td>
<td>constant</td>
</tr>
<tr>
<td>X::to_int_type(c)</td>
<td>X::int_type</td>
<td>Returns: some value e, constrained by the definitions of to_char_type and eq_int_type.</td>
<td>constant</td>
</tr>
<tr>
<td>X::eq_int_type(e,f)</td>
<td>bool</td>
<td>Returns: for all c and d, X::eq(c,d) is equal to X::eq_int_type(X::to_int_type(c), X::to_int_type(d)); otherwise, yields true if e and f are both copies of X::eof(); otherwise, yields false if one of e and f is a copy of X::eof() and the other is not; otherwise the value is unspecified.</td>
<td>constant</td>
</tr>
</tbody>
</table>
Table 69: Character traits requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::eof()</code></td>
<td><code>X::int_type</code></td>
<td>Returns a value e such that <code>X::eq_int_type(e,X::to_int_type(c))</code> is false for all values c.</td>
<td>constant</td>
</tr>
</tbody>
</table>

The class template

```
template<class charT> struct char_traits;
```

is provided in the header `<string>` as a basis for explicit specializations.

### 21.2.3 Traits typedefs

Using `int_type = see below;`

1. **Preconditions**: `int_type` shall be able to represent all of the valid characters converted from the corresponding `char_type` values, as well as an end-of-file value, `eof()`.

Using `state_type = see below;`

2. **Preconditions**: `state_type` meets the Cpp17Destructible (Table 32), Cpp17CopyAssignable (Table 31), Cpp17CopyConstructible (Table 29), and Cpp17DefaultConstructible (Table 27) requirements.

### 21.2.4 char_traits specializations

#### 21.2.4.1 General

```
namespace std {
    template<> struct char_traits<char> {
        using char_type = char;
        using int_type = int;
        using off_type = streamoff;
        using pos_type = streampos;
        using state_type = mbstate_t;
        using comparison_category = strong_ordering;
    
    static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;
    static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
    static constexpr size_t length(const char_type* s);
    static constexpr const char_type* find(const char_type* s, size_t n, const char_type& a);
    static constexpr char_type* move(char_type* s1, const char_type* s2, size_t n);
    static constexpr char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static constexpr char_type* assign(char_type* s, size_t n, char_type a);
}
```

1. The header `<string>` defines five specializations of the class template `char_traits`: `char_traits<char>`, `char_traits<char8_t>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`. If `eof()` can be held in `char_type` then some iostreams operations can give surprising results.

---

227) If `eof()` can be held in `char_type` then some iostreams operations can give surprising results.
The type `mbstate_t` is defined in `<cwchar>` and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

The two-argument member `assign` is defined identically to the built-in operator `=`. The two-argument members `eq` and `lt` are defined identically to the built-in operators `==` and `<` for type `unsigned char`.

The member `eof()` returns EOF.

---

21.2.4.3 `struct char_traits<char8_t>`

```cpp
namespace std {
  template<> struct char_traits<char8_t> {
    using char_type = char8_t;
    using int_type = unsigned int;
    using off_type = streamoff;
    using pos_type = u8streampos;
    using state_type = mbstate_t;
    using comparison_category = strong_ordering;

    static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;
    static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
    static constexpr size_t length(const char_type* s);
    static constexpr const char_type* find(const char_type* s, size_t n, const char_type& a);
    static constexpr char_type* move(char_type* s1, const char_type* s2, size_t n);
    static constexpr char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static constexpr char_type* assign(char_type* s, size_t n, char_type a);
    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
  }
}
```

---

1 The two-argument members `assign`, `eq`, and `lt` are defined identically to the built-in operators `=`, `==`, and `<` respectively.

2 The member `eof()` returns an implementation-defined constant that cannot appear as a valid UTF-8 code unit.

21.2.4.4 `struct char_traits<char16_t>`

```cpp
namespace std {
  template<> struct char_traits<char16_t> {
    using char_type = char16_t;
    using int_type = uint_least16_t;
    using off_type = streamoff;
    using pos_type = u16streampos;
    using state_type = mbstate_t;
    using comparison_category = strong_ordering;

    static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;
    static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
    static constexpr size_t length(const char_type* s);
    static constexpr const char_type* find(const char_type* s, size_t n, const char_type& a);
    static constexpr char_type* move(char_type* s1, const char_type* s2, size_t n);
    static constexpr char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static constexpr char_type* assign(char_type* s, size_t n, char_type a);
    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
  }
}
```
The two-argument members `assign`, `eq`, and `lt` are defined identically to the built-in operators `=`, `==`, and `<`, respectively.

The member `eof()` returns an implementation-defined constant that cannot appear as a valid UTF-16 code unit.

21.2.4.5 **struct char_traits<char32_t>**

```cpp
namespace std {
    template<> struct char_traits<char32_t> {
        using char_type = char32_t;
        using int_type = uint_least32_t;
        using off_type = streamoff;
        using pos_type = u32streampos;
        using state_type = mbstate_t;
        using comparison_category = strong_ordering;

        static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
        static constexpr bool eq(char_type c1, char_type c2) noexcept;
        static constexpr bool lt(char_type c1, char_type c2) noexcept;
        static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
        static constexpr size_t length(const char_type* s);
        static constexpr const char_type* find(const char_type* s, size_t n,
                                               const char_type& a);
        static constexpr char_type* move(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* copy(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* assign(char_type* s, size_t n, char_type a);

        static constexpr int_type not_eof(int_type c) noexcept;
        static constexpr char_type to_char_type(int_type c) noexcept;
        static constexpr int_type to_int_type(char_type c) noexcept;
        static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
        static constexpr int_type eof() noexcept;
    };
}
```

The two-argument members `assign`, `eq`, and `lt` are defined identically to the built-in operators `=`, `==`, and `<`, respectively.

21.2.4.6 **struct char_traits<wchar_t>**

```cpp
namespace std {
    template<> struct char_traits<wchar_t> {
        using char_type = wchar_t;
        using int_type = wint_t;
        using off_type = streamoff;
        using pos_type = wstreampos;
        using state_type = mbstate_t;

        static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
        static constexpr bool eq(char_type c1, char_type c2) noexcept;
        static constexpr bool lt(char_type c1, char_type c2) noexcept;
        static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
        static constexpr size_t length(const char_type* s);
        static constexpr const char_type* find(const char_type* s, size_t n,
                                               const char_type& a);
        static constexpr char_type* move(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* copy(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* assign(char_type* s, size_t n, char_type a);

        static constexpr int_type not_eof(int_type c) noexcept;
        static constexpr char_type to_char_type(int_type c) noexcept;
        static constexpr int_type to_int_type(char_type c) noexcept;
        static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
        static constexpr int_type eof() noexcept;
    };
}
```

The member `eof()` returns an implementation-defined constant that cannot appear as a Unicode code point.
using comparison_category = strong_ordering;

static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
static constexpr bool eq(char_type c1, char_type c2) noexcept;
static constexpr bool lt(char_type c1, char_type c2) noexcept;
static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
static constexpr size_t length(const char_type* s);
static constexpr const char_type* find(const char_type* s, size_t n, const char_type& a);
static constexpr char_type* move(char_type* s1, const char_type* s2, size_t n);
static constexpr char_type* copy(char_type* s1, const char_type* s2, size_t n);
static constexpr char_type* assign(char_type* s, size_t n, char_type a);
static constexpr int not_eof(int_type c) noexcept;
static constexpr char_type to_char_type(int_type c) noexcept;
static constexpr int_type to_int_type(char_type c) noexcept;
static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
static constexpr int_type eof() noexcept;
};

1 The two-argument members assign, eq, and lt are defined identically to the built-in operators =, ==, and <, respectively.

2 The member eof() returns WEOF.

21.3 String classes

21.3.1 General

The header <string> defines the basic_string class template for manipulating varying-length sequences of char-like objects and five typedef-names, string, u8string, u16string, u32string, and wstring, that name the specializations basic_string<char>, basic_string<char8_t>, basic_string<char16_t>, basic_string<char32_t>, and basic_string<wchar_t>, respectively.

21.3.2 Header <string> synopsis

#include <compare>    // see 17.11.1
#include <initializer_list>  // see 17.10.2

namespace std {

// 21.3.3, basic_string
template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
class basic_string;

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator> operator+(const basic_string<charT, traits, Allocator>& lhs, const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator> operator+(basic_string<charT, traits, Allocator>&& lhs, const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator> operator+(const basic_string<charT, traits, Allocator>& lhs, basic_string<charT, traits, Allocator>&& rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator> operator+(basic_string<charT, traits, Allocator>&& lhs, basic_string<charT, traits, Allocator>&& rhs);

§ 21.3.2
template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(basic_string<charT, traits, Allocator>&& lhs,
    basic_string<charT, traits, Allocator>&& rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const charT* lhs,
    const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>& lhs,
    const charT* rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(charT lhs,
    const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(charT lhs,
    basic_string<charT, traits, Allocator>&& rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>& lhs,
    const charT* rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(basic_string<charT, traits, Allocator>&& lhs,
    const charT* rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>& lhs,
    charT rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(basic_string<charT, traits, Allocator>&& lhs,
    charT rhs);

template<class charT, class traits, class Allocator>
constexpr bool
operator==(const basic_string<charT, traits, Allocator>& lhs,
    const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
constexpr bool
operator==(const basic_string<charT, traits, Allocator>& lhs,
    const charT* rhs);

template<class charT, class traits, class Allocator>
constexpr <see below>
operator<=>(const basic_string<charT, traits, Allocator>& lhs,
    const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
constexpr <see below>
operator<=>(const basic_string<charT, traits, Allocator>& lhs,
    const charT* rhs);

template<class charT, class traits, class Allocator>
void
swap(basic_string<charT, traits, Allocator>& lhs,
    basic_string<charT, traits, Allocator>& rhs) noexcept
    (noexcept(lhs.swap(rhs)));

// 21.3.4.3, swapper
template<class charT, class traits, class Allocator>
void
swap(basic_string<charT, traits, Allocator>& lhs,
    basic_string<charT, traits, Allocator>& rhs) noexcept
    (noexcept(lhs.swap(rhs)));

// 21.3.4.4, inserters and extractors
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>&& is,
    basic_string<charT, traits, Allocator>&& str);
template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const basic_string<charT, traits, Allocator>& str);

template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
getline(basic_istream<charT, traits>&& is, basic_string<charT, traits, Allocator>& str, charT delim);

// 21.3.4.5, erasure
template<class charT, class traits, class Allocator, class U>
constexpr typename basic_string<charT, traits, Allocator>::size_type
erase(basic_string<charT, traits, Allocator>& c, const U& value);

template<class charT, class traits, class Allocator, class Predicate>
constexpr typename basic_string<charT, traits, Allocator>::size_type
erase_if(basic_string<charT, traits, Allocator>& c, Predicate pred);

using string = basic_string<char>;
using u8string = basic_string<char8_t>;
using u16string = basic_string<char16_t>;
using u32string = basic_string<char32_t>;
using wstring = basic_string<wchar_t>;

// 21.3.5, numeric conversions
int stoi(const string& str, size_t* idx = nullptr, int base = 10);
long stol(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const string& str, size_t* idx = nullptr, int base = 10);
long long stoll(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = nullptr, int base = 10);
float stof(const string& str, size_t* idx = nullptr);
double stod(const string& str, size_t* idx = nullptr);
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(double val);
string to_string(long double val);

int stoi(const wstring& str, size_t* idx = nullptr, int base = 10);
long stol(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = nullptr, int base = 10);
long long stoll(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = nullptr, int base = 10);
float stof(const wstring& str, size_t* idx = nullptr);
double stod(const wstring& str, size_t* idx = nullptr);
long double stold(const wstring& str, size_t* idx = nullptr);
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

namespace pmr {
    template<class charT, class traits = char_traits<charT>>
        using basic_string = std::basic_string<charT, traits, polymorphic_allocator<charT>>;
    using string = basic_string<char>;
    using u8string = basic_string<char8_t>;
    using u16string = basic_string<char16_t>;
    using u32string = basic_string<char32_t>;
    using wstring = basic_string<wchar_t>;
}

// 21.3.6, hash support
template<class T> struct hash;
template<> struct hash<string>;
template<> struct hash<u8string>;
template<> struct hash<u16string>;
template<> struct hash<u32string>;
template<> struct hash<wstring>;
template<> struct hash<pmr::string>;
template<> struct hash<pmr::u8string>;
template<> struct hash<pmr::u16string>;
template<> struct hash<pmr::u32string>;
template<> struct hash<pmr::wstring>;

inline namespace literals {
    inline namespace string_literals {
        constexpr string operator"s(const char* str, size_t len);
        constexpr u8string operator"s(const char8_t* str, size_t len);
        constexpr u16string operator"s(const char16_t* str, size_t len);
        constexpr u32string operator"s(const char32_t* str, size_t len);
        constexpr wstring operator"s(const wchar_t* str, size_t len);
    }
}

21.3.3 Class template basic_string

The class template basic_string describes objects that can store a sequence consisting of a varying number of arbitrary char-like objects with the first element of the sequence at position zero. Such a sequence is also called a “string” if the type of the char-like objects that it holds is clear from context. In the rest of 21.3.3, the type of the char-like objects held in a basic_string object is designated by charT.

A specialization of basic_string is a contiguous container (22.2.1).

In all cases, [data(), data() + size()] is a valid range, data() + size() points at an object with value charT() (a “null terminator”), and size() <= capacity() is true.

namespace std {
    template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT>>
        class basic_string {
        public:

§ 21.3.3.1 768
// types
using traits_type = traits;
using value_type = charT;
using allocator_type = Allocator;
using size_type = typename allocator_traits<Allocator>::size_type;
using difference_type = typename allocator_traits<Allocator>::difference_type;
using pointer = typename allocator_traits<Allocator>::pointer;
using const_pointer = typename allocator_traits<Allocator>::const_pointer;
using reference = value_type&;
using const_reference = const value_type&;
using iterator = implementation_defined; // see 22.2
using const_iterator = implementation_defined; // see 22.2
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;
static const size_type npos = -1;

// 21.3.3.3, construct/copy/destroy
constexpr basic_string() noexcept(noexcept(Allocator())) : basic_string(Allocator()) { }
constexpr explicit basic_string(const Allocator& a) noexcept;
constexpr basic_string(const basic_string& str);
constexpr basic_string(basic_string&& str) noexcept;
constexpr basic_string(const basic_string& str, size_type pos,
    const Allocator& a = Allocator());
constexpr basic_string(const basic_string& str, size_type pos, size_type n,
    const Allocator& a = Allocator());
template<class T>
    constexpr basic_string(const T& t, size_type pos, size_type n,
        const Allocator& a = Allocator());
template<class T>
    constexpr explicit basic_string(const T& t, const Allocator& a = Allocator());
constexpr basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
constexpr basic_string(const charT* s, const Allocator& a = Allocator());
constexpr basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
    constexpr basic_string(InputIterator begin, InputIterator end,
        const Allocator& a = Allocator());
constexpr basic_string(initializer_list<charT>, const Allocator& = Allocator());
constexpr basic_string(const basic_string&, const Allocator&);
constexpr basic_string(basic_string&&, const Allocator&);
constexpr basic_string& operator=(const basic_string& str);
constexpr basic_string& operator=(basic_string&& str)
    noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
    allocator_traits<Allocator>::is_always_equal::value);
template<class T>
    constexpr basic_string& operator=(const T& t);
constexpr basic_string& operator=(const charT* s);
constexpr basic_string& operator=(charT c);
constexpr basic_string& operator=(initializer_list<charT>);

// 21.3.3.4, iterators
constexpr iterator begin() noexcept;
constexpr const_iterator begin() const noexcept;
constexpr iterator end() noexcept;
constexpr const_iterator end() const noexcept;
constexpr reverse_iterator rbegin() noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator rend() noexcept;
constexpr const_reverse_iterator rend() const noexcept;

§ 21.3.3.1 769
```cpp
constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

// 21.3.3.5, capacity
constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;
constexpr size_type max_size() const noexcept;
constexpr void resize(size_type n, charT c);
constexpr void resize(size_type n);
constexpr size_type capacity() const noexcept;
constexpr void reserve(size_type res_arg);
constexpr void shrink_to_fit();
constexpr void clear() noexcept;
[[nodiscard]] constexpr bool empty() const noexcept;

// 21.3.3.6, element access
constexpr const_reference operator[](size_type pos) const;
constexpr reference operator[](size_type pos);
constexpr const_reference at(size_type n) const;
constexpr reference at(size_type n);
constexpr const charT& front() const;
constexpr charT& front();
constexpr const charT& back() const;
constexpr charT& back();

// 21.3.3.7, modifiers
constexpr basic_string& operator+=(const basic_string& str);
template<class T>
    constexpr basic_string& operator+=(const T& t);
constexpr basic_string& operator+=(const charT* s);
constexpr basic_string& operator+=(charT c);
constexpr basic_string& append(const basic_string& str);
constexpr basic_string& append(const basic_string& str, size_type pos, size_type n = npos);
template<class T>
    constexpr basic_string& append(const T& t);
template<class T>
    constexpr basic_string& append(const T& t, size_type pos, size_type n = npos);
constexpr basic_string& append(const charT* s, size_type n);
constexpr basic_string& append(const charT* s);
constexpr basic_string& append(size_type n, charT c);
template<class InputIterator>
    constexpr basic_string& append(InputIterator first, InputIterator last);
constexpr basic_string& append(initializer_list<charT>);
constexpr void push_back(charT c);
constexpr basic_string& assign(const basic_string& str);
constexpr basic_string& assign(basic_string&& str)
    noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
    allocator_traits<Allocator>::is_always_equal::value);
constexpr basic_string& assign(const basic_string& str, size_type pos, size_type n = npos);
template<class T>
    constexpr basic_string& assign(const T& t);
template<class T>
    constexpr basic_string& assign(const T& t, size_type pos, size_type n = npos);
constexpr basic_string& assign(const charT* s, size_type n);
constexpr basic_string& assign(const charT* s);
constexpr basic_string& assign(size_type n, charT c);
template<class InputIterator>
    constexpr basic_string& assign(InputIterator first, InputIterator last);
```
constexpr basic_string& assign(initializer_list<charT>);
constexpr basic_string& insert(size_type pos, const basic_string& str);
constexpr basic_string& insert(size_type pos1, const basic_string& str,
    size_type pos2, size_type n = npos);
template<class T>
constexpr basic_string& insert(size_type pos, const T& t);
template<class T>
constexpr basic_string& insert(size_type pos1, const T& t,
    size_type pos2, size_type n = npos);
constexpr basic_string& insert(size_type pos, const charT* s, size_type n);
constexpr basic_string& insert(size_type pos, const charT* s);
constexpr basic_string& insert(size_type pos, size_type n, charT c);
constexpr iterator insert(const_iterator p, charT c);
constexpr iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
constexpr iterator insert(const_iterator p, InputIterator first, InputIterator last);
constexpr iterator insert(const_iterator p, initializer_list<charT>);
constexpr basic_string& erase(size_type pos = 0, size_type n = npos);
constexpr iterator erase(const_iterator p);
constexpr iterator erase(const_iterator first, const_iterator last);
constexpr void pop_back();
constexpr basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
constexpr basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
    size_type pos2, size_type n2 = npos);
template<class T>
constexpr basic_string& replace(size_type pos1, size_type n1, const T& t);
template<class T>
constexpr basic_string& replace(size_type pos1, size_type n1, const T& t,
    size_type pos2, size_type n2 = npos);
constexpr basic_string& replace(size_type pos, size_type n1, const charT* s, size_type n2);
constexpr basic_string& replace(size_type pos, size_type n1, const charT* s);
constexpr basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
constexpr basic_string& replace(const_iterator i1, const_iterator i2,
    const basic_string& str);
template<class T>
constexpr basic_string& replace(const_iterator i1, const_iterator i2, const T& t);
constexpr basic_string& replace(const_iterator i1, const_iterator i2, const charT* s,
    size_type n);
constexpr basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);
template<class InputIterator>
constexpr basic_string& replace(const_iterator i1, const_iterator i2,
    InputIterator j1, InputIterator j2);
constexpr basic_string& replace(const_iterator, const_iterator, initializer_list<charT>);
constexpr size_type copy(charT* s, size_type n, size_type pos = 0) const;
constexpr void swap(basic_string& str)
    noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
    allocator_traits<Allocator>::is_always_equal::value);

// 21.3.3.8, string operations
constexpr const charT* c_str() const noexcept;
constexpr const charT* data() const noexcept;
constexpr charT* data() noexcept;
constexpr operator basic_string_view<charT, traits>() const noexcept;
constexpr allocator_type get_allocator() const noexcept;

template<class T>
constexpr size_type find(const T& t, size_type pos = 0) const noexcept(see below);
constexpr size_type find(const basic_string& str, size_type pos = 0) const noexcept;
constexpr size_type find(const charT* s, size_type pos, size_type n) const;
constexpr size_type find(const charT* s, size_type pos = 0) const;
constexpr size_type find(charT c, size_type pos = 0) const noexcept;

template<class T>
constexpr size_type rfind(const T& t, size_type pos = npos) const noexcept;
constexpr size_type rfind(const basic_string& str, size_type pos = npos) const noexcept;
constexpr size_type rfind(const charT* s, size_type pos, size_type n) const;
constexpr size_type rfind(const charT* s, size_type pos = npos) const;
constexpr size_type rfind(charT c, size_type pos = npos) const noexcept;

template<class T>
constexpr size_type rfind_first_of(const T& t, size_type pos = 0) const noexcept;
constexpr size_type rfind_first_of(const basic_string& str, size_type pos = 0) const noexcept;
constexpr size_type rfind_first_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type rfind_first_of(const charT* s, size_type pos = 0) const;
constexpr size_type rfind_first_of(charT c, size_type pos = 0) const noexcept;

template<class T>
constexpr size_type rfind_last_of(const T& t, size_type pos = npos) const noexcept;
constexpr size_type rfind_last_of(const basic_string& str, size_type pos = npos) const noexcept;
constexpr size_type rfind_last_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type rfind_last_of(const charT* s, size_type pos = npos) const;
constexpr size_type rfind_last_of(charT c, size_type pos = npos) const noexcept;

constexpr basic_string substr(size_type pos = 0, size_type n = npos) const;

template<class T>
constexpr int compare(const T& t) const noexcept;

template<class T>
constexpr int compare(size_type pos1, size_type n1, const T& t) const;
template<class T>
constexpr int compare(size_type pos1, size_type n1, const T& t, size_type pos2, size_type n2 = npos) const;

constexpr int compare(const basic_string& str) const;
constexpr int compare(size_type pos1, size_type n1, const basic_string& str) const;
constexpr int compare(size_type pos1, size_type n1, const basic_string& str, size_type pos2, size_type n2 = npos) const;

constexpr bool starts_with(basic_string_view<charT, traits> x) const noexcept;
constexpr bool starts_with(charT x) const noexcept;
constexpr bool starts_with(charT* x) const;
constexpr bool ends_with(basic_string_view<charT, traits> x) const noexcept;
constexpr bool ends_with(charT x) const noexcept;
constexpr bool ends_with(const charT* x) const;

template<class InputIterator,
         class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
basic_string(InputIterator, InputIterator, Allocator = Allocator())
  -> basic_string<typename iterator_traits<InputIterator>::value_type,
                  char_traits<typename iterator_traits<InputIterator>::value_type>,
                  Allocator>;

template<class charT,
         class traits,
         class Allocator = allocator<charT>>
explicit basic_string(basic_string_view<charT, traits>, const Allocator& = Allocator())
  -> basic_string<charT, traits, Allocator>;

template<class charT,
         class traits,
         class Allocator = allocator<charT>>
basic_string(basic_string_view<charT, traits>,
                  typename see below::size_type, typename see below::size_type,
                  const Allocator& = Allocator())
  -> basic_string<charT, traits, Allocator>;

A size_type parameter type in a basic_string deduction guide refers to the size_type member type of
the type deduced by the deduction guide.

21.3.3.2 General requirements [string.require]

1 If any operation would cause size() to exceed max_size(), that operation throws an exception object of
type length_error.

2 If any member function or operator of basic_string throws an exception, that function or operator has no
other effect on the basic_string object.

3 In every specialization basic_string<charT, traits, Allocator>, the type allocator_traits<Alloc-
ator>::value_type shall name the same type as charT. Every object of type basic_string<charT,
traits, Allocator> uses an object of type Allocator to allocate and free storage for the contained
charT objects as needed. The Allocator object used is obtained as described in 22.2.1. In every spe-
cialization basic_string<charT, traits, Allocator>, the type traits shall meet the character traits
requirements (21.2).

[Note 1: The program is ill-formed if traits::char_type is not the same type as charT. — end note]

4 References, pointers, and iterators referring to the elements of a basic_string sequence may be invalidated
by the following uses of that basic_string object:

(4.1) — Passing as an argument to any standard library function taking a reference to non-const basic_string
as an argument.228

(4.2) — Calling non-const member functions, except operator[], at, data, front, back, begin, rbegin, end,
and rend.

21.3.3.3 Constructors and assignment operators [string.cons]

constexpr explicit basic_string(const Allocator& a) noexcept;

Postconditions: size() is equal to 0.

constexpr basic_string(const basic_string& str);
constexpr basic_string(basic_string&& str) noexcept;

Effects: Constructs an object whose value is that of str prior to this call.

Remarks: In the second form, str is left in a valid but unspecified state.

228) For example, as an argument to non-member functions swap() (21.3.4.3), operator>>() (21.3.4.4), and getline() (21.3.4.4),
or as an argument to basic_string::swap().
constexpr basic_string(const basic_string& str, size_type pos, const Allocator& a = Allocator());

**Effects:** Let n be npos for the first overload. Equivalent to:
```
basic_string(basic_string_view<charT, traits>(str).substr(pos, n), a)
```

```template<class T>
constexpr basic_string(const T& t, size_type pos, size_type n, const Allocator& a = Allocator());
```

**Constraints:**
- is_convertible_v<const T&, basic_string_view<charT, traits>> is true.

**Effects:** Creates a variable, sv, as if by
```
basic_string_view<charT, traits> sv = t;
```
and then behaves the same as:
```
basic_string(sv.substr(pos, n), a);
```

```template<class T>
constexpr explicit basic_string(const T& t, const Allocator& a = Allocator());
```

**Constraints:**
- (7.1) is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
- (7.2) is_convertible_v<const T&, const charT*> is false.

**Effects:** Creates a variable, sv, as if by
```
basic_string_view<charT, traits> sv = t;
```
and then behaves the same as
```
basic_string(sv.data(), sv.size(), a);
```

```constexpr basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
```

**Preconditions:** [s, s + n) is a valid range.

**Effects:** Constructs an object whose initial value is the range [s, s + n).

```constexpr basic_string(const charT* s, const Allocator& a = Allocator());
```

**Constraints:** Allocator is a type that qualifies as an allocator (22.2.1).

**Note 1:** This affects class template argument deduction.  

**Effects:** Equivalent to:
```
basic_string(s, traits::length(s), a);
```

```constexpr basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

**Constraints:** Allocator is a type that qualifies as an allocator (22.2.1).

**Note 2:** This affects class template argument deduction.  

**Effects:** Constructs an object whose value consists of n copies of c.

```template<class InputIterator>
constexpr basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());
```

**Constraints:** InputIterator is a type that qualifies as an input iterator (22.2.1).

**Effects:** Constructs a string from the values in the range [begin, end), as indicated in Table 77.

```constexpr basic_string(initializer_list<charT> il, const Allocator& a = Allocator());
```

**Effects:** Equivalent to basic_string(il.begin(), il.end(), a).

```constexpr basic_string(const basic_string& str, const Allocator& alloc);
constexpr basic_string(basic_string&& str, const Allocator& alloc);
```

**Effects:** Constructs an object whose value is that of str prior to this call. The stored allocator is constructed from alloc. In the second form, str is left in a valid but unspecified state.

**Throws:** The second form throws nothing if alloc == str.get_allocator().

```template<class InputIterator, 
    class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
basic_string(InputIterator, InputIterator, Allocator = Allocator())
-> basic_string<typename iterator_traits<InputIterator>::value_type,
```
char_traits<typename iterator_traits<InputIterator>::value_type>,
      Allocator>;

Constraints: InputIterator is a type that qualifies as an input iterator, and Allocator is a type that
qualifies as an allocator (22.2.1).

template<class charT,
   class traits,
   class Allocator = allocator<charT>>
explicit basic_string(basic_string_view<charT, traits>, const Allocator& = Allocator())
    -> basic_string<charT, traits, Allocator>;

Constraints: Allocator is a type that qualifies as an allocator (22.2.1).

castexpr basic_string& operator=(const basic_string& str);

Effects: If *this and str are the same object, has no effect. Otherwise, replaces the value of *this
with a copy of str.

Returns: *this.

castexpr basic_string& operator=(basic_string&& str)
noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
           allocator_traits<Allocator>::is_always_equal::value);

Effects: Move assigns as a sequence container (22.2), except that iterators, pointers and references may
be invalidated.

Returns: *this.

template<class T>
   castexpr basic_string& operator=(const T& t);

Constraints:
(27.1) — is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
(27.2) — is_convertible_v<const T&, const charT*> is false.

Effects: Equivalent to:

   basic_string_view<charT, traits> sv = t;
   return assign(sv);

   castexpr basic_string& operator=(const charT* s);

Effects: Equivalent to: return *this = basic_string_view<charT, traits>(s);

   castexpr basic_string& operator=(charT c);

Effects: Equivalent to:

   return *this = basic_string_view<charT, traits>(addressof(c), 1);

   castexpr basic_string& operator=(initializer_list<charT> il);

Effects: Equivalent to:

   return *this = basic_string_view<charT, traits>(il.begin(), il.size());

21.3.3.4 Iterator support

   castexpr iterator begin() noexcept;
   castexpr const_iterator begin() const noexcept;
   castexpr const_iterator cbegin() const noexcept;

   Returns: An iterator referring to the first character in the string,
constexpr iterator end() noexcept;
constexpr const_iterator end() const noexcept;
constexpr const_iterator cend() const noexcept;

Returns: An iterator which is the past-the-end value.

constexpr reverse_iterator rbegin() noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;

Returns: An iterator which is semantically equivalent to reverse_iterator(end()).

constexpr reverse_iterator rend() noexcept;
constexpr const_reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

Returns: An iterator which is semantically equivalent to reverse_iterator(begin()).

21.3.3.5 Capacity

constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;

Returns: A count of the number of char-like objects currently in the string.
Complexity: Constant time.

constexpr size_type max_size() const noexcept;

Returns: The largest possible number of char-like objects that can be stored in a basic_string.
Complexity: Constant time.

constexpr void resize(size_type n, charT c);

Effects: Alters the value of *this as follows:
- If n <= size(), erases the last size() - n elements.
- If n > size(), appends n - size() copies of c.

constexpr void resize(size_type n);

Effects: Equivalent to resize(n, charT()).

constexpr size_type capacity() const noexcept;

Returns: The size of the allocated storage in the string.
Complexity: Constant time.

constexpr void reserve(size_type res_arg);

Effects: A directive that informs a basic_string of a planned change in size, so that the storage allocation can be managed accordingly. After reserve(), capacity() is greater or equal to the argument of reserve if reallocation happens; and equal to the previous value of capacity() otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of reserve().

Throws: length_error if res_arg > max_size() or any exceptions thrown by allocator_traits<Allocator>::allocate.

constexpr void shrink_to_fit();

Effects: shrink_to_fit is a non-binding request to reduce capacity() to size().
[Note 1: The request is non-binding to allow latitude for implementation-specific optimizations. — end note]
It does not increase capacity(), but may reduce capacity() by causing reallocation.

Complexity: If the size is not equal to the old capacity, linear in the size of the sequence; otherwise constant.

Remarks: Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence, as well as the past-the-end iterator.
21.3.3.6 Element access

constexpr const_reference operator[](size_type pos) const;
constexpr reference operator[](size_type pos);

Preconditions: pos <= size().

Returns: *(begin() + pos) if pos < size(). Otherwise, returns a reference to an object of type charT with value charT(), where modifying the object to any value other than charT() leads to undefined behavior.

Throws: Nothing.

Complexity: Constant time.

constexpr const_reference at(size_type pos) const;
constexpr reference at(size_type pos);

Returns: operator[](pos).

Throws: out_of_range if pos >= size().

constexpr const charT& front() const;
constexpr charT& front();

Preconditions: !empty().

Effects: Equivalent to: return operator[](0);

constexpr const charT& back() const;
constexpr charT& back();

Preconditions: !empty().

Effects: Equivalent to: return operator[](size() - 1);

§ 21.3.3.7 Modifiers

21.3.3.7.1 basic_string::operator+=

constexpr basic_string& operator+=(const basic_string& str);

Effects: Equivalent to: return append(str);

template<class T>
constexpr basic_string& operator+=(const T& t);

Constraints:
(2.1) is_convertible_v<const T&, basic_string_view<charT, traits>> is true and

(2.2) is_convertible_v<const T&, const charT*> is false.

Effects: Equivalent to:

basic_string_view<charT, traits> sv = t;
return append(sv);

constexpr basic_string& operator+=(const charT* s);
Effects: Equivalent to: return append(s);

constexpr basic_string& operator+=(charT c);
Effects: Equivalent to: return append(size_type{1}, c);
constexpr basic_string& operator+=(initializer_list<charT> il);

Effects: Equivalent to: return append(il);

21.3.3.7.2 basic_string::append

constexpr basic_string& append(const basic_string& str);
Effects: Equivalent to: return append(str.data(), str.size());

constexpr basic_string& append(const basic_string& str, size_type pos, size_type n = npos);
Effects: Equivalent to:
return append(basic_string_view<charT, traits>(str).substr(pos, n));

template<class T>
constexpr basic_string& append(const T& t);

Effects: Equivalent to:
basic_string_view<charT, traits> sv = t;
return append(sv.data(), sv.size());

template<class T>
constexpr basic_string& append(const T& t, size_type pos, size_type n = npos);

Effects: Equivalent to:
basic_string_view<charT, traits> sv = t;
return append(sv.substr(pos, n));

constexpr basic_string& append(const charT* s, size_type n);
Preconditions: [s, s + n) is a valid range.
Effects: Appends a copy of the range [s, s + n) to the string.
Returns: *this.

constexpr basic_string& append(const charT* s);
Effects: Equivalent to: return append(s, traits::length(s));

constexpr basic_string& append(size_type n, charT c);
Effects: Appends n copies of c to the string.
Returns: *this.

template<class InputIterator>
constexpr basic_string& append(InputIterator first, InputIterator last);

Constraints: InputIterator is a type that qualifies as an input iterator (22.2.1).
Effects: Equivalent to: return append(basic_string(first, last, get_allocator()));

constexpr basic_string& append(initializer_list<charT> il);
Effects: Equivalent to: return append(il.begin(), il.size());

constexpr void push_back(charT c);
Effects: Equivalent to append(size_type{1}, c).
21.3.3.7.3 basic_string::assign

```cpp
constexpr basic_string& assign(const basic_string& str);

**Effects:** Equivalent to: return *this = str;
```

```cpp
constexpr basic_string& assign(const basic_string& str)
  noexcept(allocation_traits<Allocator>::propagate_on_container_move_assignment::value ||
            allocation_traits<Allocator>::is_always_equal::value);
```

```cpp
Effects: Equivalent to: return *this = std::move(str);
```

```cpp
constexpr basic_string& assign(const basic_string& str, size_type pos, size_type n = npos);

**Effects:** Equivalent to:
return assign(basic_string_view<charT, traits>(str).substr(pos, n));
```

```cpp
template<class T>
constexpr basic_string& assign(const T& t);

**Constraints:**
— (4.1) is_convertible_v<const T&, basic_string_view<CharT, traits>> is true and
— (4.2) is_convertible_v<const T&, const CharT*> is false.
```

```cpp
Effects: Equivalent to:
basic_string_view<CharT, traits> sv = t;
return assign(sv.data(), sv.size());
```

```cpp
template<class T>
constexpr basic_string& assign(const T& t, size_type pos, size_type n = npos);

**Constraints:**
— (6.1) is_convertible_v<const T&, basic_string_view<CharT, traits>> is true and
— (6.2) is_convertible_v<const T&, const CharT*> is false.
```

```cpp
Effects: Equivalent to:
basic_string_view<CharT, traits> sv = t;
return assign(sv.substr(pos, n));
```

```cpp
constexpr basic_string& assign(const charT* s, size_type n);

**Preconditions:** [s, s + n) is a valid range.
**Effects:** Replaces the string controlled by *this with a copy of the range [s, s + n).
**Returns:** *this.
```

```cpp
constexpr basic_string& assign(const charT* s);

**Effects:** Equivalent to: return assign(s, traits::length(s));
```

```cpp
constexpr basic_string& assign(initializer_list<CharT> il);

**Effects:** Equivalent to: return assign(il.begin(), il.size());
```

```cpp
constexpr basic_string& assign(size_type n, CharT c);

**Effects:** Equivalent to:
clear();
resize(n, c);
return *this;
```

```cpp
template<class InputIterator>
constexpr basic_string& assign(InputIterator first, InputIterator last);

**Constraints:** InputIterator is a type that qualifies as an input iterator (22.2.1).

**Effects:** Equivalent to: return assign(basic_string(first, last, get_allocator()));
```
21.3.3.7.4 basic_string::insert

constexpr basic_string& insert(size_type pos, const basic_string& str);

**Effects**: Equivalent to: return insert(pos, str.data(), str.size());

constexpr basic_string& insert(size_type pos1, const basic_string& str, size_type pos2, size_type n = npos);

**Effects**: Equivalent to:

```cpp
class T
constexpr basic_string& insert(size_type pos, const T& t);
```

**Constraints**:

1. is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
2. is_convertible_v<const T&, const charT*> is false.

```cpp
template<class T>
constexpr basic_string& insert(size_type pos1, const T& t, size_type pos2, size_type n = npos);
```

**Constraints**:

1. is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
2. is_convertible_v<const T&, const charT*> is false.

**Effects**: Equivalent to:

```cpp
basic_string_view<charT, traits> sv = t;
return insert(pos1, sv.data(), sv.size());
```

```cpp
template<class T>
constexpr basic_string& insert(size_type pos, const charT* s, size_type n);
```

**Effects**: Inserts \(n\) copies of \(c\) before the character at position \(pos\) if \(pos <\) size(), or otherwise at the end of the string.

**Returns**: *this.

**Throws**:

1. out_of_range if \(pos >\) size(),
2. length_error if \(n >\) max_size() - size(), or
3. any exceptions thrown by allocator_traits<Allocator>::allocate.

```cpp
constexpr basic_string& insert(size_type pos, const charT* s);
```

**Effects**: Equivalent to: return insert(pos, s, traits::length(s));

```cpp
constexpr basic_string& insert(size_type pos, size_type n, charT c);
```

**Effects**: Inserts \(n\) copies of \(c\) before the character at position \(pos\) if \(pos <\) size(), or otherwise at the end of the string.

**Returns**: *this

**Throws**:

1. out_of_range if \(pos >\) size(),
2. length_error if \(n >\) max_size() - size(), or
3. any exceptions thrown by allocator_traits<Allocator>::allocate.
constexpr iterator insert(const_iterator p, charT c);
  
  **Preconditions:** p is a valid iterator on *this.
  
  **Effects:** Inserts a copy of c at the position p.
  
  **Returns:** An iterator which refers to the inserted character.

constexpr iterator insert(const_iterator p, size_type n, charT c);
  
  **Preconditions:** p is a valid iterator on *this.
  
  **Effects:** Inserts n copies of c at the position p.
  
  **Returns:** An iterator which refers to the first inserted character, or p if n == 0.

template<class InputIterator>
constexpr iterator insert(const_iterator p, InputIterator first, InputIterator last);
  
  **Constraints:** InputIterator is a type that qualifies as an input iterator (22.2.1).
  
  **Preconditions:** p is a valid iterator on *this.
  
  **Effects:** Equivalent to insert(p - begin(), basic_string(first, last, get_allocator())).
  
  **Returns:** An iterator which refers to the first inserted character, or p if first == last.

constexpr iterator insert(const_iterator p, initializer_list<charT> il);
  
  **Effects:** Equivalent to: return insert(p, il.begin(), il.end());

21.3.3.7.5  **basic_string::erase**  
[string.erase]

constexpr basic_string& erase(size_type pos = 0, size_type n = npos);
  
  **Effects:** Determines the effective length xlen of the string to be removed as the smaller of n and size() - pos. Removes the characters in the range [begin() + pos, begin() + pos + xlen].
  
  **Returns:** *this.
  
  **Throws:** out_of_range if pos > size().

constexpr iterator erase(const_iterator p);
  
  **Preconditions:** p is a valid dereferenceable iterator on *this.
  
  **Effects:** Removes the character referred to by p.
  
  **Returns:** An iterator which points to the element immediately following p prior to the element being erased. If no such element exists, end() is returned.
  
  **Throws:** Nothing.

constexpr iterator erase(const_iterator first, const_iterator last);
  
  **Preconditions:** first and last are valid iterators on *this. [first, last) is a valid range.
  
  **Effects:** Removes the characters in the range [first, last).
  
  **Returns:** An iterator which points to the element pointed to by last prior to the other elements being erased. If no such element exists, end() is returned.
  
  **Throws:** Nothing.

constexpr void pop_back();
  
  **Preconditions:** !empty().
  
  **Effects:** Equivalent to erase(end() - 1).
  
  **Throws:** Nothing.

21.3.3.7.6  **basic_string::replace**  
[string.replace]

constexpr basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
  
  **Effects:** Equivalent to: return replace(pos1, n1, str.data(), str.size());
constexpr basic_string& replace(size_type pos1, size_type n1, const basic_string& str, size_type pos2, size_type n2 = npos);

Effects: Equivalent to:
        return replace(pos1, n1, basic_string_view<charT, traits>(str).substr(pos2, n2));

template<class T>
constexpr basic_string& replace(size_type pos1, size_type n1, const T& t);

Effects: Equivalent to:
        basic_string_view<charT, traits> sv = t;
        return replace(pos1, n1, sv.data(), sv.size());

template<class T>
constexpr basic_string& replace(size_type pos1, size_type n1, const T& t, size_type pos2, size_type n2 = npos);

Constraints:
(3.1) is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
(3.2) is_convertible_v<const T&, const charT*> is false.

Effects: Equivalent to:
        basic_string_view<charT, traits> sv = t;
        return replace(pos1, n1, sv.substr(pos2, n2));

constexpr basic_string& replace(size_type pos1, size_type n1, const charT* s, size_type n2);

Effects: Equivalent to:
        return replace(pos1, n1, s, traits::length(s));

constexpr basic_string& replace(size_type pos1, size_type n1, size_type n2, charT c);

Effects: Determines the effective length xlen of the string to be removed as the smaller of n1 and size() - pos1. If size() - xlen >= max_size() - n2 throws length_error. Otherwise, the function replaces the characters in the range [begin() + pos1, begin() + pos1 + xlen) with n2 copies of c.

Returns: *this.

Throws:
(10.1) out_of_range if pos1 > size(),
(10.2) length_error if the length of the resulting string would exceed max_size(), or
(10.3) any exceptions thrown by allocator_traits<Allocator>::allocate.

constexpr basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);

Effects: Equivalent to: return replace(i1, i2, basic_string_view<charT, traits>(str));
template<class T>
constant basic_string& replace(const_iterator i1, const_iterator i2, const T& t);

Constraints:
— is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
— is_convertible_v<const T&, const charT*> is false.

Preconditions: [begin(), i1) and [i1, i2) are valid ranges.

Effects: Equivalent to:

constant basic_string& replace(const_iterator i1, const_iterator i2, const charT* s, size_type n);

Effects: Equivalent to:

constant basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);

Effects: Equivalent to:

constant basic_string& replace(const_iterator i1, const_iterator i2, size_type n, charT c);

Preconditions: [begin(), i1) and [i1, i2) are valid ranges.

Effects: Equivalent to:

template<class InputIterator>
constant basic_string& replace(const_iterator i1, const_iterator i2, InputIterator j1, InputIterator j2);

Constraints: InputIterator is a type that qualifies as an input iterator (22.2.1).

Effects: Equivalent to:

constant basic_string& replace(const_iterator i1, const_iterator i2, initializer_list<charT> il);

Effects: Equivalent to:

21.3.3.7.7 basic_string::copy

constexpr size_type copy(charT* s, size_type n, size_type pos = 0) const;

Effects: Equivalent to: return basic_string_view<charT, traits>(*this).copy(s, n, pos);

[Note 1: This does not terminate s with a null object. —end note]

21.3.3.7.8 basic_string::swap

constexpr void swap(basic_string& s) noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value);

Preconditions: allocator_traits<Allocator>::propagate_on_container_swap::value is true or
get_allocator() == s.get_allocator().

Postconditions: *this contains the same sequence of characters that was in s, s contains the same
sequence of characters that was in *this.

Throws: Nothing.

Complexity: Constant time.

21.3.3.8 String operations

21.3.3.8.1 Accessors

constexpr const charT* c_str() const noexcept;
constexpr const charT* data() const noexcept;

Returns: A pointer p such that p + i == addressof(operator[](i)) for each i in [0, size()].

Complexity: Constant time.
Remarks: The program shall not modify any of the values stored in the character array; otherwise, the behavior is undefined.

```cpp
constexpr charT* data() noexcept;
```

Returns: A pointer \( p \) such that \( p + i == \text{addressof}(\text{operator}[i]) \) for each \( i \) in \([0, \text{size()})\).

Complexity: Constant time.

Remarks: The program shall not modify the value stored at \( p + \text{size()} \) to any value other than \( \text{charT}() \); otherwise, the behavior is undefined.

```cpp
constexpr operator basic_string_view<charT, traits>() const noexcept;
```

Effects: Equivalent to: return basic_string_view<charT, traits>(data(), size());

```cpp
constexpr allocator_type get_allocator() const noexcept;
```

Returns: A copy of the Allocator object used to construct the string or, if that allocator has been replaced, a copy of the most recent replacement.

### 21.3.3.8.2 Searching [string.find]

Let \( F \) be one of \( \text{find}, \text{rfind}, \text{find_first_of}, \text{find_last_of}, \text{find_first_not_of}, \) and \( \text{find_last_not_of} \).

---

(1.1) — Each member function of the form

```cpp
constexpr size_type F(const basic_string& str, size_type pos) const noexcept;
```

has effects equivalent to: return \( F(\text{basic_string_view}\langle\text{charT}, \text{traits}\rangle(\text{str}), \text{pos}) \);

(1.2) — Each member function of the form

```cpp
constexpr size_type F(const charT* s, size_type pos) const;
```

has effects equivalent to: return \( F(\text{basic_string_view}\langle\text{charT}, \text{traits}\rangle(s), \text{pos}) \);

(1.3) — Each member function of the form

```cpp
constexpr size_type F(const charT* s, size_type pos, size_type n) const;
```

has effects equivalent to: return \( F(\text{basic_string_view}\langle\text{charT}, \text{traits}\rangle(s, n), \text{pos}) \);

(1.4) — Each member function of the form

```cpp
constexpr size_type F(charT c, size_type pos) const noexcept;
```

has effects equivalent to:

```cpp
return F(\text{basic_string_view}\langle\text{charT}, \text{traits}\rangle(\text{addressof}(c), 1), \text{pos}) ;
```

---

#### Constraints:

(2.1) — is_convertible_v<\const T&, basic_string_view<\charT, traits>> is true and

(2.2) — is_convertible_v<\const T&, \const charT*>> is false.

Effects: Let \( G \) be the name of the function. Equivalent to:

```cpp
basic_string_view<\charT, traits> s = *this, sv = t;
return s.G(sv, pos);
```

Remarks: The expression inside noexcept is equivalent to is_nothrow_convertible_v<\const T&, basic_string_view<\charT, traits>>.
21.3.3.8.3 basic_string::substr

```
constexpr basic_string substr(size_type pos = 0, size_type n = npos) const;
```

**Effects:** Determines the effective length \( rlen \) of the string to copy as the smaller of \( n \) and \( \text{size}() - \text{pos} \).

**Returns:** basic_string(data()+pos, rlen).

**Throws:** out_of_range if pos > size().

21.3.3.8.4 basic_string::compare

```
template<class T>
constexpr int compare(const T& t) const noexcept;
```

**Constraints:**
- (1.1) is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
- (1.2) is_convertible_v<const T&, const charT*> is false.

**Effects:** Equivalent to: return basic_string_view<charT, traits>(*this).compare(t);

**Remarks:** The expression inside noexcept is equivalent to is_nothrow_convertible_v<const T&, basic_string_view<charT, traits>>.

```
template<class T>
constexpr int compare(size_type pos1, size_type n1, const T& t) const;
```

**Constraints:**
- (4.1) is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
- (4.2) is_convertible_v<const T&, const charT*> is false.

**Effects:** Equivalent to:
```
return basic_string_view<charT, traits>(*this).substr(pos1, n1).compare(t);
```

```
template<class T>
constexpr int compare(size_type pos1, size_type n1, const T& t, size_type pos2, size_type n2 = npos) const;
```

**Constraints:**
- (6.1) is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
- (6.2) is_convertible_v<const T&, const charT*> is false.

**Effects:** Equivalent to:
```
basic_string_view<charT, traits> s = *this, sv = t;
return s.substr(pos1, n1).compare(sv.substr(pos2, n2));
```

```
constexpr int compare(const basic_string& str) const noexcept;
```

**Effects:** Equivalent to:
```
return compare(basic_string_view<charT, traits>(str));
```

```
constexpr int compare(size_type pos1, size_type n1, const basic_string& str) const;
```

**Effects:** Equivalent to:
```
return compare(pos1, n1, basic_string_view<charT, traits>(str));
```

```
constexpr int compare(size_type pos1, size_type n1, const basic_string& str, size_type pos2, size_type n2 = npos) const;
```

**Effects:** Equivalent to:
```
return compare(pos1, n1, basic_string_view<charT, traits>(str), pos2, n2);
```

```
constexpr int compare(const charT* s) const;
```

**Effects:** Equivalent to:
```
return compare(basic_string_view<charT, traits>(s));
```

```
constexpr int compare(size_type pos, size_type n, const charT* s) const;
```

**Effects:** Equivalent to:
```
return compare(pos, n, basic_string_view<charT, traits>(s));
```
constexpr int compare(size_type pos, size_type n1, const charT* s, size_type n2) const;

**Effects:** Equivalent to: \( \text{return compare(pos, n1, \text{basic_string_view}<\text{charT}, \text{traits}>(s, n2));} \)

### 21.3.3.8.5 basic_string::starts_with

```cpp
constexpr bool starts_with(basic_string_view<charT, traits> x) const noexcept;
constexpr bool starts_with(charT x) const noexcept;
constexpr bool starts_with(const charT* x) const;
```

**Effects:** Equivalent to:

```cpp
\text{return basic_string_view<\text{charT}, \text{traits}>(data(), size()).starts_with(x);} \)
```

### 21.3.3.8.6 basic_string::ends_with

```cpp
constexpr bool ends_with(basic_string_view<charT, traits> x) const noexcept;
constexpr bool ends_with(charT x) const noexcept;
constexpr bool ends_with(const charT* x) const;
```

**Effects:** Equivalent to:

```cpp
\text{return basic_string_view<\text{charT}, \text{traits}>(data(), size()).ends_with(x);} \)
```

### 21.3.4 Non-member functions

#### 21.3.4.1 operator+

```cpp
template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>& lhs,
           const basic_string<charT, traits, Allocator>& rhs);
```

**Effects:** Equivalent to:

```cpp
\text{basic_string<\text{charT}, \text{traits}, \text{Allocator}> r = \text{lhs};}
\text{r.append(rhs);} \)
\text{return r;} \)
```

```cpp
template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>&\& lhs,
           const basic_string<charT, traits, Allocator>&\& rhs);
```

**Effects:** Equivalent to:

```cpp
\text{lhs.append(rhs);} \)
\text{return std::move(lhs);} \)
```

```cpp
template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(basic_string<charT, traits, Allocator>&& lhs,
           basic_string<charT, traits, Allocator>&& rhs);
```

**Effects:** Equivalent to:

```cpp
\text{lhs.append(rhs);} \)
\text{return std::move(lhs);} \)
```

except that both \( \text{lhs} \) and \( \text{rhs} \) are left in valid but unspecified states.

**Note 1:** If \( \text{lhs} \) and \( \text{rhs} \) have equal allocators, the implementation can move from either. — end note

```cpp
template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>& lhs,
           basic_string<charT, traits, Allocator>&\& rhs);
```

```cpp
template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(basic_string<charT, traits, Allocator>&\& lhs,
           const basic_string<charT, traits, Allocator>& rhs);
```

```cpp
template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(basic_string<charT, traits, Allocator>&\& lhs,
           basic_string<charT, traits, Allocator>& rhs);
```

```cpp
template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>&\& lhs,
           basic_string<charT, traits, Allocator>& rhs);
```
template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const charT* lhs, basic_string<charT, traits, Allocator>&& rhs);

Effects: Equivalent to:
    rhs.insert(0, lhs);
    return std::move(rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

Effects: Equivalent to:
    basic_string<charT, traits, Allocator> r = rhs;
    r.insert(0, lhs);
    return r;

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(charT lhs, const basic_string<charT, traits, Allocator>& rhs);

Effects: Equivalent to:
    basic_string<charT, traits, Allocator> r = rhs;
    r.insert(r.begin(), lhs);
    return r;

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(charT lhs, basic_string<charT, traits, Allocator>&& rhs);

Effects: Equivalent to:
    rhs.insert(rhs.begin(), lhs);
    return std::move(rhs);

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>& lhs, charT rhs);

Effects: Equivalent to:
    basic_string<charT, traits, Allocator> r = lhs;
    r.push_back(rhs);
    return r;

template<class charT, class traits, class Allocator>
constexpr basic_string<charT, traits, Allocator>
operator+(basic_string<charT, traits, Allocator>&& lhs, charT rhs);

Effects: Equivalent to:
    lhs.push_back(rhs);
    return std::move(lhs);

21.3.4.2 Non-member comparison operator functions [string.cmp]

template<class charT, class traits, class Allocator>
constexpr bool
operator==(const basic_string<charT, traits, Allocator>& lhs,
            const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
constexpr bool
operator==(const basic_string<charT, traits, Allocator>& lhs,
            const charT* rhs);

template<class charT, class traits, class Allocator>
constexpr see below
operator<=>(const basic_string<charT, traits, Allocator>& lhs,
            const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
constexpr see below operator<=>(const basic_string<charT, traits, Allocator>& lhs,
const charT* rhs);
1
Effects: Let op be the operator. Equivalent to:
return basic_string_view<charT, traits>(lhs) op basic_string_view<charT, traits>(rhs);

21.3.4.3 swap [string.special]

template<class charT, class traits, class Allocator>
constexpr void
swap(basic_string<charT, traits, Allocator>& lhs,
 basic_string<charT, traits, Allocator>& rhs)
noexcept(noexcept(lhs.swap(rhs)));
1
Effects: Equivalent to lhs.swap(rhs).

21.3.4.4 Inserters and extractors [string.io]

template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>&& is, basic_string<charT, traits, Allocator>& str);
1
Effects: Behaves as a formatted input function (29.7.4.3.1). After constructing a sentry object, if the
sentry converts to true, calls str.erase() and then extracts characters from is and appends them to
str as if by calling str.append(1, c). If is.width() is greater than zero, the maximum number n
of characters appended is is.width(); otherwise n is str.max_size(). Characters are extracted and
appended until any of the following occurs:
1. n characters are stored;
2. end-of-file occurs on the input sequence;
3. isspace(c, is.getloc()) is true for the next available input character c.

After the last character (if any) is extracted, is.width(0) is called and the sentry object is destroyed.
If the function extracts no characters, it calls is.setstate(ios_base::failbit), which may throw
ios_base::failure (29.5.5.4).

Returns: is.

template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
 const basic_string<charT, traits, Allocator>& str);
5
Effects: Equivalent to: return os << basic_string_view<charT, traits>(str);

template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
getline(basic_istream<charT, traits>& is, basic_string<charT, traits, Allocator>& str,
 charT delim);

Effects: Behaves as an unformatted input function (29.7.4.4), except that it does not affect the value
returned by subsequent calls to basic_istream<>::gcount(). After constructing a sentry object,
if the sentry converts to true, calls str.erase() and then extracts characters from is and appends
them to str as if by calling str.append(1, c) until any of the following occurs:
1. end-of-file occurs on the input sequence (in which case, the getline function calls is.setstate(
   ios_base::eofbit)).
2. traits::eq(c, delim) for the next available input character c (in which case, c is extracted but
   not appended) (29.5.5.4)
str.max_size() characters are stored (in which case, the function calls is.setstate(ios_base::failbit)) (29.5.5.4)

The conditions are tested in the order shown. In any case, after the last character is extracted, the sentry object is destroyed.

If the function extracts no characters, it calls is.setstate(ios_base::failbit) which may throw ios_base::failure (29.5.5.4).

Returns: is.

template<class charT, class traits, class Allocator>
    basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>& is,
            basic_string<charT, traits, Allocator>& str);

template<class charT, class traits, class Allocator>
    basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
            basic_string<charT, traits, Allocator>& str);

Returns: getline(is, str, is.widen('
')).

21.3.4.5 Erasure

template<class charT, class traits, class Allocator, class U>
    constexpr typename basic_string<charT, traits, Allocator>::size_type
    erase(basic_string<charT, traits, Allocator>& c, const U& value);

Effects: Equivalent to:
    auto it = remove(c.begin(), c.end(), value);
    auto r = distance(it, c.end());
    c.erase(it, c.end());
    return r;

template<class charT, class traits, class Allocator, class Predicate>
    constexpr typename basic_string<charT, traits, Allocator>::size_type
    erase_if(basic_string<charT, traits, Allocator>& c, Predicate pred);

Effects: Equivalent to:
    auto it = remove_if(c.begin(), c.end(), pred);
    auto r = distance(it, c.end());
    c.erase(it, c.end());
    return r;

21.3.5 Numeric conversions

int stoi(const string& str, size_t* idx = nullptr, int base = 10);
long stol(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const string& str, size_t* idx = nullptr, int base = 10);
long long stoll(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = nullptr, int base = 10);

Effects: The first two functions call strtol(str.c_str(), ptr, base), and the last three functions call stoul(str.c_str(), ptr, base), strtoll(str.c_str(), ptr, base), and strtoull(str.c_str(), ptr, base), respectively. Each function returns the converted result, if any. The argument ptr designates a pointer to an object internal to the function that is used to determine what to store at *idx. If the function does not throw an exception and idx != nullptr, the function stores in *idx the index of the first unconverted element of str.

Returns: The converted result.

Throws: invalid_argument if strtol, stoul, stoll, or strtoull reports that no conversion could be performed. Throws out_of_range if strtol, stoul, stoll or strtoull sets errno to ERANGE, or if the converted value is outside the range of representable values for the return type.

float stof(const string& str, size_t* idx = nullptr);
double stod(const string& str, size_t* idx = nullptr);
long double stold(const string& str, size_t* idx = nullptr);

Effects: These functions call `strtof(str.c_str(), ptr)`, `strtod(str.c_str(), ptr)`, and `strtold(str.c_str(), ptr)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != nullptr`, the function stores in `*idx` the index of the first unconverted element of `str`.

Returns: The converted result.

Throws: `invalid_argument` if `strtof`, `strtod`, or `strtold` reports that no conversion could be performed. Throws `out_of_range` if `strtof`, `strtod`, or `strtold` sets `errno` to `ERANGE` or if the converted value is outside the range of representable values for the return type.

string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);

Returns: Each function returns a `string` object holding the character representation of the value of its argument that would be generated by calling `sprintf(buf, fmt, val)` with a format specifier of "%d", "%u", "%ld", "%lu", "%lld", "%llu", "%f", or "%Lf", respectively, where `buf` designates an internal character buffer of sufficient size.

int stoi(const wstring& str, size_t* idx = nullptr, int base = 10);
long stol(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = nullptr, int base = 10);
long long stoll(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = nullptr, int base = 10);

Effects: The first two functions call `wcstol(str.c_str(), ptr, base)`, and the last three functions call `wcstoul(str.c_str(), ptr, base)`, `wcstoll(str.c_str(), ptr, base)`, and `wcstoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != nullptr`, the function stores in `*idx` the index of the first unconverted element of `str`.

Returns: The converted result.

Throws: `invalid_argument` if `wcstol`, `wcstoul`, `wcstoll`, or `wcstoull` reports that no conversion could be performed. Throws `out_of_range` if the converted value is outside the range of representable values for the return type.

float stof(const wstring& str, size_t* idx = nullptr);
double stod(const wstring& str, size_t* idx = nullptr);
long double stold(const wstring& str, size_t* idx = nullptr);

Effects: These functions call `wcstof(str.c_str(), ptr)`, `wcstod(str.c_str(), ptr)`, and `wcstold(str.c_str(), ptr)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != nullptr`, the function stores in `*idx` the index of the first unconverted element of `str`.

Returns: The converted result.

Throws: `invalid_argument` if `wcstof`, `wcstod`, or `wcstold` reports that no conversion could be performed. Throws `out_of_range` if `wcstof`, `wcstod`, or `wcstold` sets `errno` to `ERANGE`.

wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);

§ 21.3.5
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

Returns: Each function returns a wstring object holding the character representation of the value of its argument that would be generated by calling swprintf(buf, buffsz, fmt, val) with a format specifier of L"%d", L"%u", L"%ld", L"%lu", L"%lld", L"%llu", L"%f", L"%f", or L"%Lf", respectively, where buf designates an internal character buffer of sufficient size buffsz.

21.3.6 Hash support

template<> struct hash<string>;
template<> struct hash<u8string>;
template<> struct hash<u16string>;
template<> struct hash<u32string>;
template<> struct hash<wstring>;
template<> struct hash<pmr::string>;
template<> struct hash<pmr::u8string>;
template<> struct hash<pmr::u16string>;
template<> struct hash<pmr::u32string>;
template<> struct hash<pmr::wstring>;

1 If S is one of these string types, SV is the corresponding string view type, and s is an object of type S, then hash<S>()(s) == hash<SV>()(SV(s)).

21.3.7 Suffix for basic_string literals

constexpr string operator"s(const char* str, size_t len);
Returns: string{str, len}.

constexpr u8string operator"s(const char8_t* str, size_t len);
Returns: u8string{str, len}.

constexpr u16string operator"s(const char16_t* str, size_t len);
Returns: u16string{str, len}.

constexpr u32string operator"s(const char32_t* str, size_t len);
Returns: u32string{str, len}.

constexpr wstring operator"s(const wchar_t* str, size_t len);
Returns: wstring{str, len}.

6 [Note 1: The same suffix s is used for chrono::duration literals denoting seconds but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. —end note]

21.4 String view classes

21.4.1 General

The class template basic_string_view describes an object that can refer to a constant contiguous sequence of char-like (21.1) objects with the first element of the sequence at position zero. In the rest of 21.4, the type of the char-like objects held in a basic_string_view object is designated by charT.

[Note 1: The library provides implicit conversions from const char* and std::basic_string<charT, ...> to std::basic_string_view<charT, ...> so that user code can accept just std::basic_string_view<charT> as a non-templated parameter wherever a sequence of characters is expected. User-defined types can define their own implicit conversions to std::basic_string_view in order to interoperate with these functions. —end note]

21.4.2 Header <string_view> synopsis

#include <compare>  // see 17.11.1

namespace std {
  // 21.4.3, class template basic_string_view
  template<class charT, class traits = char_traits<charT>>
    class basic_string_view;
The function templates defined in 20.2.2 and 23.7 are available when `<string_view>` is included.

### 21.4.3 Class template `basic_string_view`

#### 21.4.3.1 General

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_string_view {
        public:
            // types
            using traits_type = traits;
            using value_type = charT;
            using pointer = value_type*;
            using const_pointer = const value_type*;
            using reference = value_type&;
            using const_reference = const value_type&;
            using const_iterator = implementation-defined; // see 21.4.3.3
```
using iterator = const_iterator;
using const_reverse_iterator = reverse_iterator<const_iterator>;
using reverse_iterator = const_reverse_iterator;
using size_type = size_t;
using difference_type = ptrdiff_t;
static constexpr size_type npos = size_type(-1);

// 21.4.3.2, construction and assignment
constexpr basic_string_view() noexcept;
constexpr basic_string_view(const basic_string_view&) noexcept = default;
constexpr basic_string_view& operator=(const basic_string_view&) noexcept = default;
constexpr basic_string_view(const charT* str);
constexpr basic_string_view(const charT* str, size_type len);
template<class It, class End>
  constexpr basic_string_view(It begin, End end);

// 21.4.3.3, iterator support
constexpr const_iterator begin() const noexcept;
constexpr const_iterator end() const noexcept;
constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr const_reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

// 21.4.3.4, capacity
constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;
constexpr size_type max_size() const noexcept;
[[nodiscard]] constexpr bool empty() const noexcept;

// 21.4.3.5, element access
constexpr const_reference operator[](size_type pos) const;
constexpr const_reference at(size_type pos) const;
constexpr const_reference front() const;
constexpr const_reference back() const;
constexpr const_pointer data() const noexcept;

// 21.4.3.6, modifiers
constexpr void remove_prefix(size_type n);
constexpr void remove_suffix(size_type n);
constexpr void swap(basic_string_view& s) noexcept;

// 21.4.3.7, string operations
constexpr size_type copy(charT* s, size_type n, size_type pos = 0) const;
constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const;

constexpr int compare(basic_string_view s) const noexcept;
constexpr int compare(size_type pos1, size_type n1, basic_string_view s) const;
constexpr int compare(size_type pos1, size_type n1, basic_string_view s, size_type pos2, size_type n2) const;
constexpr int compare(const charT* s) const;
constexpr int compare(size_type pos1, size_type n1, const charT* s) const;
constexpr int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;
constexpr bool starts_with(basic_string_view x) const noexcept;
constexpr bool starts_with(charT x) const noexcept;
constexpr bool starts_with(const charT* x) const;
constexpr bool ends_with(basic_string_view x) const noexcept;
constexpr bool ends_with(charT x) const noexcept;

229) Because basic_string_view refers to a constant sequence, iterator and const_iterator are the same type.
constexpr bool ends_with(const charT* x) const;

// 21.4.3.8, searching
constexpr size_type find(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find(charT c, size_type pos = 0) const noexcept;
constexpr size_type find(const charT* s, size_type pos, size_type n) const;
constexpr size_type find(const charT* s, size_type pos = 0) const;
constexpr size_type rfind(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type rfind(charT c, size_type pos = npos) const noexcept;
constexpr size_type rfind(const charT* s, size_type pos, size_type n) const;
constexpr size_type rfind(const charT* s, size_type pos = npos) const;
constexpr size_type find_first_of(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find_first_of(charT c, size_type pos = 0) const noexcept;
constexpr size_type find_first_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_first_of(const charT* s, size_type pos = 0) const;
constexpr size_type find_last_of(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type find_last_of(charT c, size_type pos = npos) const noexcept;
constexpr size_type find_last_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_last_of(const charT* s, size_type pos = npos) const;
constexpr size_type find_first_not_of(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
constexpr size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_first_not_of(const charT* s, size_type pos = 0) const;
constexpr size_type find_last_not_of(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type find_last_not_of(charT c, size_type pos = npos) const noexcept;
constexpr size_type find_last_not_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_last_not_of(const charT* s, size_type pos = npos) const;

private:
    const_pointer data_; // exposition only
    size_type size_; // exposition only
};

// 21.4.4, deduction guide
template<class It, class End>
    basic_string_view(It, End) -> basic_string_view<iter_value_t<It>>;

1 In every specialization basic_string_view<charT, traits>, the type traits shall meet the character traits requirements (21.2).
[Note 1: The program is ill-formed if traits::char_type is not the same type as charT. — end note]

2 For a basic_string_view str, any operation that invalidates a pointer in the range [str.data(), str.data() + str.size()) invalidates pointers, iterators, and references returned from str’s member functions.

3 The complexity of basic_string_view member functions is $O(1)$ unless otherwise specified.

21.4.3.2 Construction and assignment

constexpr basic_string_view() noexcept;
1 Postconditions: size_ == 0 and data_ == nullptr.

constexpr basic_string_view(const charT* str);
2 Preconditions: [str, str + traits::length(str)) is a valid range.
3 Effects: Constructs a basic_string_view, initializing data_ with str and initializing size_ with traits::length(str).
4 Complexity: $O$(traits::length(str)).

§ 21.4.3.2 794
constexpr basic_string_view(const charT* str, size_type len);

Preconditions: 
[ str, str + len ) is a valid range.

Effects: Constructs a basic_string_view, initializing data_ with str and initializing size_ with len.

template<class It, class End>
constexpr basic_string_view(It begin, End end);

Constraints:
(7.1) It satisfies contiguous_iterator.
(7.2) End satisfies sized_sentinel_for<It>.
(7.3) is_same_v<iter_value_t<It>, charT> is true.
(7.4) is_convertible_v<End, size_type> is false.

Preconditions:
(8.1) [ begin, end ) is a valid range.
(8.2) It models contiguous_iterator.
(8.3) End models sized_sentinel_for<It>.

Effects: Initializes data_ with to_address(begin) and initializes size_ with end - begin.

21.4.3.3 Iterator support [string.view.iterators]

using const_iterator = implementation-defined;

A type that meets the requirements of a constant Cpp17RandomAccessIterator (23.3.5.7), models contiguous_iterator (23.3.4.14), and meets the constexpr iterator requirements (23.3.1), whose value_type is the template parameter charT.

All requirements on container iterators (22.2) apply to basic_string_view::const_iterator as well.

constexpr const_iterator begin() const noexcept;
constexpr const_iterator cbegin() const noexcept;

Returns: An iterator such that
(3.1) if !empty(), addressof(*begin()) == data_,
(3.2) otherwise, an unspecified value such that [ begin(), end() ) is a valid range.

constexpr const_iterator end() const noexcept;
constexpr const_iterator cend() const noexcept;

Returns: begin() + size().

constexpr const_reverse_iterator rbegin() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;

Returns: const_reverse_iterator(end()).

constexpr const_reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

Returns: const_reverse_iterator(begin()).

21.4.3.4 Capacity [string.view.capacity]

constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;

Returns: size_.

constexpr size_type max_size() const noexcept;

Returns: The largest possible number of char-like objects that can be referred to by a basic_string_view.

[[nodiscard]] constexpr bool empty() const noexcept;

Returns: size_ == 0.
21.4.3.5  Element access
[string.view.access]
constexpr const_reference operator[](size_type pos) const;
1  Preconditions: pos < size().
2  Returns: data_[pos].
3  Throws: Nothing.
4  [Note 1: Unlike basic_string::operator[], basic_string_view::operator[](size()) has undefined behavior instead of returning charT(). — end note]

constexpr const_reference at(size_type pos) const;
5  Returns: data_[pos].
6  Throws: out_of_range if pos >= size().

constexpr const_reference front() const;
7  Preconditions: !empty().
8  Returns: data_[0].
9  Throws: Nothing.

constexpr const_reference back() const;
10  Preconditions: !empty().
11  Returns: data_[size() - 1].
12  Throws: Nothing.

constexpr const_pointer data() const noexcept;
13  Returns: data_.
14  [Note 2: Unlike basic_string::data() and string-literals, data() can return a pointer to a buffer that is not null-terminated. Therefore it is typically a mistake to pass data() to a function that takes just a const charT* and expects a null-terminated string. — end note]

21.4.3.6  Modifiers
[string.view.modifiers]
constexpr void remove_prefix(size_type n);
1  Preconditions: n <= size().
2  Effects: Equivalent to: data_ += n; size_ -= n;

castexpr void remove_suffix(size_type n);
3  Preconditions: n <= size().
4  Effects: Equivalent to: size_ -= n;

castexpr void swap(basic_string_view& s) noexcept;
5  Effects: Exchanges the values of *this and s.

21.4.3.7  String operations
[string.view.ops]
castexpr size_type copy(charT* s, size_type n, size_type pos = 0) const;
1  Let rlen be the smaller of n and size() - pos.
2  Preconditions: [s, s + rlen) is a valid range.
3  Effects: Equivalent to traits::copy(s, data() + pos, rlen).
4  Returns: rlen.
5  Throws: out_of_range if pos > size().
6  Complexity: ϴ(rlen).

constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const;
7  Let rlen be the smaller of n and size() - pos.
Effects: Determines $rlen$, the effective length of the string to reference.

Returns: basic_string_view(data() + pos, rlen).

Throws: out_of_range if $pos > size()$. 

constexpr int compare(basic_string_view str) const noexcept;

Let $rlen$ be the smaller of $size()$ and str.size().

Effects: Determines $rlen$, the effective length of the strings to compare. The function then compares the two strings by calling traits::compare(data(), str.data(), rlen).

Returns: The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 70.

Table 70: compare() results  

<table>
<thead>
<tr>
<th>Condition</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$size() &lt; str.size()</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>$size() == str.size()</td>
<td>0</td>
</tr>
<tr>
<td>$size() &gt; str.size()</td>
<td>&gt; 0</td>
</tr>
</tbody>
</table>

Complexity: $O(rlen)$.

constexpr int compare(size_type pos1, size_type n1, basic_string_view str) const;

Effects: Equivalent to: return substr(pos1, n1).compare(str);

constexpr int compare(size_type pos1, size_type n1, basic_string_view str, size_type pos2, size_type n2) const;

Effects: Equivalent to: return substr(pos1, n1).compare(str.substr(pos2, n2));

constexpr int compare(const charT* s) const;

Effects: Equivalent to: return compare(basic_string_view(s));

constexpr int compare(size_type pos1, size_type n1, const charT* s) const;

Effects: Equivalent to: return substr(pos1, n1).compare(basic_string_view(s));

constexpr int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;

Effects: Equivalent to: return substr(pos1, n1).compare(basic_string_view(s, n2));

constexpr bool starts_with(basic_string_view x) const noexcept;

Effects: Equivalent to: return substr(0, x.size()) == x;

constexpr bool starts_with(charT x) const noexcept;

Effects: Equivalent to: return !empty() && traits::eq(front(), x);

constexpr bool starts_with(const charT* x) const;

Effects: Equivalent to: return starts_with(basic_string_view(x));

constexpr bool ends_with(basic_string_view x) const noexcept;

Effects: Equivalent to:
return size() >= x.size() && compare(size() - x.size(), npos, x) == 0;

constexpr bool ends_with(charT x) const noexcept;

Effects: Equivalent to: return !empty() && traits::eq(back(), x);

constexpr bool ends_with(const charT* x) const;

Effects: Equivalent to: return ends_with(basic_string_view(x));
21.4.3.8 Searching  

Member functions in this subclause have complexity $O(\text{size()} \times \text{str.size()})$ at worst, although implementations should do better.

Let $F$ be one of `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, and `find_last_not_of`.

(2.1) Each member function of the form

```cpp
    constexpr return-type F(const charT* s, size_type pos) const;
```

has effects equivalent to:

```cpp
    return F(basic_string_view(s), pos);
```

(2.2) Each member function of the form

```cpp
    constexpr return-type F(const charT* s, size_type pos, size_type n) const;
```

has effects equivalent to:

```cpp
    return F(basic_string_view(s, n), pos);
```

(2.3) Each member function of the form

```cpp
    constexpr return-type F(charT c, size_type pos) const noexcept;
```

has effects equivalent to:

```cpp
    return F(basic_string_view(addressof(c), 1), pos);
```

```cpp
    constexpr size_type find(basic_string_view str, size_type pos = 0) const noexcept;
```

Let $\text{xpos}$ be the lowest position, if possible, such that the following conditions hold:

(3.1) $\text{pos} \leq \text{xpos}$

(3.2) $\text{xpos} + \text{str.size()} \leq \text{size()}$

(3.3) $\text{traits::eq(at(\text{xpos} + I), str.at(I))}$ for all elements $I$ of the string referenced by $\text{str}$.

Effects: Determines $\text{xpos}$.

Returns: $\text{xpos}$ if the function can determine such a value for $\text{xpos}$. Otherwise, returns $\text{npos}$.

```cpp
    constexpr size_type rfind(basic_string_view str, size_type pos = npos) const noexcept;
```

Let $\text{xpos}$ be the highest position, if possible, such that the following conditions hold:

(6.1) $\text{xpos} \leq \text{pos}$

(6.2) $\text{xpos} + \text{str.size()} \leq \text{size()}$

(6.3) $\text{traits::eq(at(\text{xpos} + I), str.at(I))}$ for all elements $I$ of the string referenced by $\text{str}$.

Effects: Determines $\text{xpos}$.

Returns: $\text{xpos}$ if the function can determine such a value for $\text{xpos}$. Otherwise, returns $\text{npos}$.

```cpp
    constexpr size_type find_first_of(basic_string_view str, size_type pos = 0) const noexcept;
```

Let $\text{xpos}$ be the lowest position, if possible, such that the following conditions hold:

(9.1) $\text{pos} \leq \text{xpos}$

(9.2) $\text{xpos} < \text{size()}$

(9.3) $\text{traits::eq(at(\text{xpos}), str.at(I))}$ for some element $I$ of the string referenced by $\text{str}$.

Effects: Determines $\text{xpos}$.

Returns: $\text{xpos}$ if the function can determine such a value for $\text{xpos}$. Otherwise, returns $\text{npos}$.

```cpp
    constexpr size_type find_last_of(basic_string_view str, size_type pos = npos) const noexcept;
```

Let $\text{xpos}$ be the highest position, if possible, such that the following conditions hold:

(12.1) $\text{xpos} \leq \text{pos}$

(12.2) $\text{xpos} < \text{size()}$

(12.3) $\text{traits::eq(at(\text{xpos}), str.at(I))}$ for some element $I$ of the string referenced by $\text{str}$.

Effects: Determines $\text{xpos}$.

Returns: $\text{xpos}$ if the function can determine such a value for $\text{xpos}$. Otherwise, returns $\text{npos}$.

```cpp
    constexpr size_type find_first_not_of(basic_string_view str, size_type pos = 0) const noexcept;
```

Let $\text{xpos}$ be the lowest position, if possible, such that the following conditions hold:
— pos <= xpos
— xpos < size()
— traits::eq(at(xpos), str.at(I)) for no element I of the string referenced by str.

**Effects:** Determines xpos.

**Returns:** xpos if the function can determine such a value for xpos. Otherwise, returns npos.

```cpp
constexpr size_type find_last_not_of(basic_string_view str, size_type pos = npos) const noexcept;
```

**Let** xpos **be the highest position, if possible, such that the following conditions hold:**

— xpos <= pos
— xpos < size()
— traits::eq(at(xpos), str.at(I)) for no element I of the string referenced by str.

**Effects:** Determines xpos.

**Returns:** xpos if the function can determine such a value for xpos. Otherwise, returns npos.

### 21.4.4 Deduction guide

```cpp
template<class It, class End>
basic_string_view(It, End) -> basic_string_view<iter_value_t<It>>;
```

**Constraints:**

— It satisfies contiguous_iterator.
— End satisfies sized_sentinel_for<It>.

### 21.4.5 Non-member comparison functions

**Let** S **be** basic_string_view<charT, traits>, **and** sv **be an instance of** S. **Implementations shall provide sufficient additional overloads marked constexpr and noexcept so that an object** t **with an implicit conversion to** S **can be compared according to Table 71.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>t == sv</td>
<td>S(t) == sv</td>
</tr>
<tr>
<td>sv == t</td>
<td>sv == S(t)</td>
</tr>
<tr>
<td>t != sv</td>
<td>S(t) != sv</td>
</tr>
<tr>
<td>sv != t</td>
<td>sv != S(t)</td>
</tr>
<tr>
<td>t &lt; sv</td>
<td>S(t) &lt; sv</td>
</tr>
<tr>
<td>sv &lt; t</td>
<td>sv &lt; S(t)</td>
</tr>
<tr>
<td>t &gt; sv</td>
<td>S(t) &gt; sv</td>
</tr>
<tr>
<td>sv &gt; t</td>
<td>sv &gt; S(t)</td>
</tr>
<tr>
<td>t &lt;= sv</td>
<td>S(t) &lt;= sv</td>
</tr>
<tr>
<td>sv &lt;= t</td>
<td>sv &lt;= S(t)</td>
</tr>
<tr>
<td>t &gt;= sv</td>
<td>S(t) &gt;= sv</td>
</tr>
<tr>
<td>sv &gt;= t</td>
<td>sv &gt;= S(t)</td>
</tr>
<tr>
<td>t &lt;=&gt; sv</td>
<td>S(t) &lt;=&gt; sv</td>
</tr>
<tr>
<td>sv &lt;=&gt; t</td>
<td>sv &lt;=&gt; S(t)</td>
</tr>
</tbody>
</table>

**Example 1:** A sample conforming implementation for operator== would be:

```cpp
template<class charT, class traits>
constexpr bool operator==(basic_string_view<charT, traits> lhs, basic_string_view<charT, traits> rhs) noexcept {
    return lhs.compare(rhs) == 0;
}
```

```cpp
template<class charT, class traits>
constexpr bool operator==(basic_string_view<charT, traits> lhs, type_identity_t<basic_string_view<charT, traits>> rhs) noexcept {
    return lhs.compare(rhs) == 0;
}
```
template<class charT, class traits>
constexpr bool operator==(basic_string_view<charT, traits> lhs,
basic_string_view<charT, traits> rhs) noexcept;

Returns: lhs.compare(rhs) == 0.

template<class charT, class traits>
constexpr sees below operator<=>(basic_string_view<charT, traits> lhs,
basic_string_view<charT, traits> rhs) noexcept;

Let R denote the type traits::comparison_category if it exists, otherwise R is weak_ordering.

Returns: static_cast<R>(lhs.compare(rhs) <=> 0).

21.4.6 Inserters and extractors

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, basic_string_view<charT, traits> str);

Effects: Behaves as a formatted output function (29.7.5.3.1) of os. Forms a character sequence
seq, initially consisting of the elements defined by the range [str.begin(), str.end()). Determines
padding for seq as described in 29.7.5.3.1. Then inserts seq as if by calling
os.rdbuf()->sputn(seq, n), where n is the larger of os.width() and str.size(); then calls os.width(0).

Returns: os

21.4.7 Hash support

The specialization is enabled (20.14.19).

[Note 1: The hash value of a string view object is equal to the hash value of the corresponding string
object (21.3.6). — end note]

21.4.8 Suffix for basic_string_view literals

constexpr string_view operator"sv(const char* str, size_t len) noexcept;

Returns: string_view{str, len}.

custom u8string_view operator"sv(const char8_t* str, size_t len) noexcept;

Returns: u8string_view{str, len}.

custom u16string_view operator"sv(const char16_t* str, size_t len) noexcept;

Returns: u16string_view{str, len}.

custom u32string_view operator"sv(const char32_t* str, size_t len) noexcept;

Returns: u32string_view{str, len}.

custom wstring_view operator"sv(const wchar_t* str, size_t len) noexcept;

Returns: wstring_view{str, len}.

21.5 Null-terminated sequence utilities

21.5.1 Header <cctype> synopsis

namespace std {
  int isalnum(int c);
  int isalpha(int c);
  int isblank(int c);
}
The contents and meaning of the header `<cctype>` are the same as the C standard library header `<ctype.h>`.

See also: ISO C 7.4

### 21.5.2 Header `<cwctype>` synopsis

```cpp
namespace std {
    using wint_t = see below;
    using wctrans_t = see below;
    using wctype_t = see below;

    int iswalnum(wint_t wc);
    int iswalpha(wint_t wc);
    int iswblank(wint_t wc);
    int iswcntrl(wint_t wc);
    int iswdigit(wint_t wc);
    int iswgraph(wint_t wc);
    int iswlower(wint_t wc);
    int iswprint(wint_t wc);
    int iswpunct(wint_t wc);
    int iswspace(wint_t wc);
    int iswupper(wint_t wc);
    int iswxdigit(wint_t wc);
    int iswctype(wint_t wc, wctype_t desc);
    wctype_t wctype(const char* property);
    wint_t towlower(wint_t wc);
    wint_t towupper(wint_t wc);
    wint_t towctrans(wint_t wc, wctrans_t desc);
    wctrans_t wctrans(const char* property);
}
```

#define WEOF see below

The contents and meaning of the header `<cwctype>` are the same as the C standard library header `<wctype.h>`.

See also: ISO C 7.30

### 21.5.3 Header `<cstring>` synopsis

```cpp
namespace std {
    using size_t = see 17.2.4;

    void* memcpy(void* s1, const void* s2, size_t n);
    void* memmove(void* s1, const void* s2, size_t n);
    char* strcpy(char* s1, const char* s2);
    char* strncpy(char* s1, const char* s2, size_t n);
    char* strcat(char* s1, const char* s2);
    int memcmp(const void* s1, const void* s2, size_t n);
    int strcmp(const char* s1, const char* s2);
    int strcoll(const char* s1, const char* s2);
    int strncmp(const char* s1, const char* s2, size_t n);
    size_t strxfrm(char* s1, const char* s2, size_t n);
    const void* memchr(const void* s, int c, size_t n);
    // see 16.2
}
```
The contents and meaning of the header `<cstring>` are the same as the C standard library header `<string.h>`.

The functions `strerror` and `strtok` are not required to avoid data races (16.4.6.10).

The functions `memcpy` and `memmove` are signal-safe (17.13.5). Both functions implicitly create objects (6.7.2) in the destination region of storage immediately prior to copying the sequence of characters to the destination.

[Note 1: The functions `strchr`, `strpbrk`, `strrchr`, `strstr`, and `memchr`, have different signatures in this document, but they have the same behavior as in the C standard library (16.2). — end note]

See also: ISO C 7.24

### 21.5.4 Header `<cwchar>` synopsis

```cpp
namespace std {
    using size_t = see 17.2.4;
    using mbstate_t = see below;
    using wint_t = see below;

    struct tm;

    int fprintf(FILE* stream, const wchar_t* format, ...);
    int fscanf(FILE* stream, const wchar_t* format, ...);
    int swprintf(wchar_t* s, size_t n, const wchar_t* format, ...);
    int sscanf(const wchar_t* s, const wchar_t* format, ...);
    int vfprintf(FILE* stream, const wchar_t* format, va_list arg);
    int vfscanf(FILE* stream, const wchar_t* format, va_list arg);
    int vwprintf(const wchar_t* format, va_list arg);
    int vwscanf(const wchar_t* format, va_list arg);
    int wprintf(const wchar_t* format, ...);
    int wscanf(const wchar_t* format, ...);
    wint_t fgetwc(FILE* stream);
    wchar_t* fgetws(wchar_t* s, int n, FILE* stream);
    wint_t fputwc(wchar_t c, FILE* stream);
    int fputws(const wchar_t* s, FILE* stream);
    int fwide(FILE* stream, int mode);
    wint_t getwc(FILE* stream);
    wint_t getwchar();
    wint_t putwc(wchar_t c, FILE* stream);
    wint_t putwchar(wchar_t c);
    int fwide(FILE* stream, int mode);
    wint_t getwc(FILE* stream);
    int fgetwc(FILE* stream);
    int fputwc(FILE* stream);
    int wputwc(FILE* stream);
    wint_t ungetwc(wchar_t c);
    double wcstod(const wchar_t* nptr, wchar_t** endptr);
    float wcstof(const wchar_t* nptr, wchar_t** endptr);
    long double wcstold(const wchar_t* nptr, wchar_t** endptr);
    long int wcstol(const wchar_t* nptr, wchar_t** endptr, int base);
    long long int wcstoll(const wchar_t* nptr, wchar_t** endptr, int base);
}
```


```c
unsigned long int wcstoul(const wchar_t* nptr, wchar_t** endptr, int base);
unsigned long long int wcstoull(const wchar_t* nptr, wchar_t** endptr, int base);
wchar_t* wcscpy(wchar_t* s1, const wchar_t* s2);
wchar_t* wcsncpy(wchar_t* s1, const wchar_t* s2, size_t n);
wchar_t* wmemcpy(wchar_t* s1, const wchar_t* s2, size_t n);
wchar_t* wmemmove(wchar_t* s1, const wchar_t* s2, size_t n);
wchar_t* wcscat(wchar_t* s1, const wchar_t* s2);
wchar_t* wcsncat(wchar_t* s1, const wchar_t* s2, size_t n);
int wcscmp(const wchar_t* s1, const wchar_t* s2);
int wcscoll(const wchar_t* s1, const wchar_t* s2);
int wcsncmp(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(wchar_t* s1, const wchar_t* s2, size_t n);
int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcschr(wchar_t* s, wchar_t c);
// see 16.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
// see 16.2
wchar_t* wcspbrk(const wchar_t* s, const wchar_t* s2);
// see 16.2
wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcspbrk(const wchar_t* s, wchar_t c);
// see 16.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2);
// see 16.2
wchar_t* wcschr(wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s1, wchar_t c);
// see 16.2
wchar_t* wcschr(wchar_t* s1, wchar_t c);
// see 16.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
// see 16.2
wchar_t* wcspbrk(const wchar_t* s, const wchar_t* s2);
// see 16.2
wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcsrchr(wchar_t* s1, wchar_t c);
// see 16.2
size_t wcsxfrm(wchar_t* s1, const wchar_t* s2, size_t n);
int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcschr(wchar_t* s, wchar_t c);
// see 16.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
// see 16.2
wchar_t* wcspbrk(const wchar_t* s, const wchar_t* s2);
// see 16.2
wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcsrchr(wchar_t* s1, wchar_t c);
// see 16.2
size_t wcsxfrm(wchar_t* s1, const wchar_t* s2, size_t n);
int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcschr(wchar_t* s, wchar_t c);
// see 16.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
// see 16.2
wchar_t* wcspbrk(const wchar_t* s, const wchar_t* s2);
// see 16.2
wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcsrchr(wchar_t* s1, wchar_t c);
// see 16.2
size_t wcsxfrm(wchar_t* s1, const wchar_t* s2, size_t n);
int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcschr(wchar_t* s, wchar_t c);
// see 16.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
// see 16.2
wchar_t* wcspbrk(const wchar_t* s, const wchar_t* s2);
// see 16.2
wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcsrchr(wchar_t* s1, wchar_t c);
// see 16.2
size_t wcsxfrm(wchar_t* s1, const wchar_t* s2, size_t n);
int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcschr(wchar_t* s, wchar_t c);
// see 16.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
// see 16.2
wchar_t* wcspbrk(const wchar_t* s, const wchar_t* s2);
// see 16.2
wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcsrchr(wchar_t* s1, wchar_t c);
// see 16.2
size_t wcsxfrm(wchar_t* s1, const wchar_t* s2, size_t n);
int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcschr(wchar_t* s, wchar_t c);
// see 16.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
// see 16.2
wchar_t* wcspbrk(const wchar_t* s, const wchar_t* s2);
// see 16.2
wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
// see 16.2
wchar_t* wcsrchr(wchar_t* s1, wchar_t c);
// see 16.2
size_t wcsxfrm(wchar_t* s1, const wchar_t* s2, size_t n);
int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n);
```

1 The contents and meaning of the header `<cwchar>` are the same as the C standard library header `<wchar.h>`, except that it does not declare a type `wchar_t`.

2 [Note 1: The functions `wcschr`, `wcspbrk`, `wcschr`, `wcsstr`, and `wmemchr` have different signatures in this document, but they have the same behavior as in the C standard library (16.2). — end note]

SEE ALSO: ISO C 7.29

21.5.5 Header `<cuchar>` synopsis

```c
namespace std {
    using mbstate_t = see below;
    using size_t = see 17.2.4;

    size_t mbstrlen(const char* s, size_t n, mbstate_t* ps);
    size_t mbtouc8(char8_t* pc8, const char* s, size_t n, mbstate_t* ps);
    size_t c8rtomb(char* s, char8_t c8, mbstate_t* ps);
    size_t mbtoucs(const char* d, const char* s, size_t n, mbstate_t* ps);
    size_t wcrtomb(char* s, wchar_t wc, mbstate_t* ps);
    size_t mbrtoc8(char8_t* pc8, const char* s, size_t n, mbstate_t* ps);
    size_t mbrtoc16(char16_t* pc16, const char* s, size_t n, mbstate_t* ps);
    size_t c16rtomb(char* s, char16_t c16, mbstate_t* ps);
    size_t mbrtoc32(char32_t* pc32, const char* s, size_t n, mbstate_t* ps);
}
```
size_t c32rtomb(char* s, char32_t c32, mbstate_t* ps);

The contents and meaning of the header <uchar> are the same as the C standard library header <uchar.h>, except that it declares the additional mbrtoc8 and c8rtomb functions and does not declare types char16_t nor char32_t.

See also: ISO C 7.28

21.5.6 Multibyte / wide string and character conversion functions [c.mb.wcs]

1 Note 1: The headers <cstdlib> (17.2.2), <uchar> (21.5.5), and <wchar> (21.5.4) declare the functions described in this subclause. —end note

int mbsinit(const mbstate_t* ps);
int mblen(const char* s, size_t n);
size_t mbstowcs(wchar_t* pwcs, const char* s, size_t n);
size_t wcstombs(char* s, const wchar_t* pwcs, size_t n);

2 Effects: These functions have the semantics specified in the C standard library.

See also: ISO C 7.22.7.1, 7.22.8, 7.29.6.2.1

int mbtowc(wchar_t* pwc, const char* s, size_t n);
int wctomb(char* s, wchar_t wchar);

3 Effects: These functions have the semantics specified in the C standard library.

4 Remarks: Calls to these functions may introduce a data race (16.4.6.10) with other calls to the same function.

See also: ISO C 7.22.7

size_t mbrlen(const char* s, size_t n, mbstate_t* ps);
size_t mbrtowc(wchar_t* pwc, const char* s, size_t n, mbstate_t* ps);
size_t wcrtomb(char* s, wchar_t wc, mbstate_t* ps);
size_t mbstowcs(wchar_t* dst, const char** src, size_t len, mbstate_t* ps);
size_t wcsrtombs(char* dst, const wchar_t** src, size_t len, mbstate_t* ps);

5 Effects: These functions have the semantics specified in the C standard library.

6 Remarks: Calling these functions with an mbstate_t* argument that is a null pointer value may introduce a data race (16.4.6.10) with other calls to the same function with an mbstate_t* argument that is a null pointer value.

See also: ISO C 7.29.6.3

size_t mbrtoc8(char8_t* pc8, const char* s, size_t n, mbstate_t* ps);

7 Effects: If s is a null pointer, equivalent to mbrtoc8(nullptr, "", 1, ps). Otherwise, the function inspects at most n bytes beginning with the byte pointed to by s to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding UTF-8 code units and then, if pc8 is not a null pointer, stores the value of the first (or only) such code unit in the object pointed to by pc8. Subsequent calls will store successive UTF-8 code units without consuming any additional input until all the code units have been stored. If the corresponding Unicode character is U+0000, the resulting state described is the initial conversion state.

8 Returns: The first of the following that applies (given the current conversion state):

(8.1) 0, if the next n or fewer bytes complete the multibyte character that corresponds to the U+0000 Unicode character (which is the value stored).

(8.2) between 1 and n (inclusive), if the next n or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

(8.3) (size_t)(-3), if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

(8.4) (size_t)(-2), if the next n bytes contribute to an incomplete (but potentially valid) multibyte character, and all n bytes have been processed (no value is stored).
— $(\text{size}_t) (-1)$, if an encoding error occurs, in which case the next $n$ or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro EILSEQ is stored in `errno`, and the conversion state is unspecified.

```c
size_t c8rtomb(char* s, char8_t c8, mbstate_t* ps);
```

**Effects:** If `s` is a null pointer, equivalent to `c8rtomb(buf, u8'\0', ps)` where `buf` is an internal buffer. Otherwise, if `c8` completes a sequence of valid UTF-8 code units, determines the number of bytes needed to represent the multibyte character (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by `s`. At most MB_CUR_MAX bytes are stored. If the multibyte character is a null character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

**Returns:** The number of bytes stored in the array object (including any shift sequences). If `c8` does not contribute to a sequence of `char8_t` corresponding to a valid multibyte character, the value of the macro EILSEQ is stored in `errno`, $(\text{size}_t) (-1)$ is returned, and the conversion state is unspecified.

**Remarks:** Calls to `c8rtomb` with a null pointer argument for `s` may introduce a data race (16.4.6.10) with other calls to `c8rtomb` with a null pointer argument for `s`. 
22 Containers library

22.1 General

This Clause describes components that C++ programs may use to organize collections of information.

The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 72.

Table 72: Containers library summary

<table>
<thead>
<tr>
<th>Subclause Header</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.2 Requirements</td>
<td></td>
</tr>
<tr>
<td>22.3 Sequence containers</td>
<td>&lt;array&gt;, &lt;deque&gt;, &lt;forward_list&gt;, &lt;list&gt;, &lt;vector&gt;</td>
</tr>
<tr>
<td>22.4 Associative containers</td>
<td>&lt;map&gt;, &lt;set&gt;</td>
</tr>
<tr>
<td>22.5 Unordered associative containers</td>
<td>&lt;unordered_map&gt;, &lt;unordered_set&gt;</td>
</tr>
<tr>
<td>22.6 Container adaptors</td>
<td>&lt;queue&gt;, &lt;stack&gt;</td>
</tr>
<tr>
<td>22.7 Views</td>
<td>&lt;span&gt;</td>
</tr>
</tbody>
</table>

22.2 Container requirements

22.2.1 General container requirements

Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

All of the complexity requirements in this Clause are stated solely in terms of the number of operations on the contained objects.

[Example 1: The copy constructor of type vector<vector<int>> has linear complexity, even though the complexity of copying each contained vector<int> is itself linear. —end example]

For the components affected by this subclause that declare an allocator_type, objects stored in these components shall be constructed using the function allocator_traits<allocator_type>::rebind_traits<U>::construct and destroyed using the function allocator_traits<allocator_type>::rebind_traits<U>::destroy (20.10.9.3), where U is either allocator_type::value_type or an internal type used by the container. These functions are called only for the container’s element type, not for internal types used by the container.

[Note 1: This means, for example, that a node-based container might need to construct nodes containing aligned buffers and call construct to place the element into the buffer. —end note]

In Tables 73, 74, and 75 X denotes a container class containing objects of type T, a and b denote values of type X, i and j denote values of type (possibly const) X::iterator, u denotes an identifier, r denotes a non-const value of type X, and rv denotes a non-const rvalue of type X.

Table 73: Container requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::value_type T</td>
<td></td>
<td></td>
<td>Preconditions: T is Cpp17Erasable from X (see 22.2.1, below)</td>
<td>compile time</td>
</tr>
<tr>
<td>X::reference T&amp;</td>
<td></td>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::const_reference const T&amp;</td>
<td></td>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note</td>
<td>Complexity</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>----------------</td>
<td>------------</td>
</tr>
<tr>
<td>X::iterator</td>
<td>iterator type whose value type is T</td>
<td>any iterator category that meets the forward iterator requirements, convertible to X::const_iterator.</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::const_iterator</td>
<td>constant iterator type whose value type is T</td>
<td>any iterator category that meets the forward iterator requirements.</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::difference_type</td>
<td>signed integer type</td>
<td>is identical to the difference type of X::iterator and X::const_iterator</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::size_type</td>
<td>unsigned integer type</td>
<td>size_type can represent any non-negative value of difference_type</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X u;</td>
<td></td>
<td>Postconditions: u.empty()</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>X()</td>
<td></td>
<td>Postconditions: X().empty()</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>X(a)</td>
<td></td>
<td>Preconditions: T is Cpp17CopyInsertable into X (see below). Postconditions: a == X(a).</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>X u(a); X u = a;</td>
<td></td>
<td>Preconditions: T is Cpp17CopyInsertable into X (see below). Postconditions: u == a</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>X u(rv); X u = rv;</td>
<td></td>
<td>Postconditions: u is equal to the value that rv had before this construction</td>
<td>(Note B)</td>
<td></td>
</tr>
<tr>
<td>a = rv X&amp;</td>
<td>All existing elements of a are either move assigned to or destroyed</td>
<td>Postconditions: a is equal to the value that rv had before this assignment</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a.~X();</td>
<td>void</td>
<td>Effects: destroys every element of a; any memory obtained is deallocated.</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a.begin(); const_iterator for constant a</td>
<td>iterator; const_iterator for constant a</td>
<td></td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.end(); const_iterator for constant a</td>
<td>iterator; const_iterator for constant a</td>
<td></td>
<td>constant</td>
<td></td>
</tr>
</tbody>
</table>
Table 73: Container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.begin()</td>
<td>const_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a) .begin();</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.end()</td>
<td>const_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).end();</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>i &lt;=&gt; j</td>
<td>strong_ordering</td>
<td>Constraints: X::iterator meets the random access iterator requirements.</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a == b</td>
<td>convertible to bool</td>
<td>== is an equivalence relation. equal(a.begin(), a.end(), b.begin(), b.end())</td>
<td>Preconditions: T meets the Cpp17-EqualityComparable requirements</td>
<td>Constant if a.size() != b.size(), linear otherwise</td>
</tr>
<tr>
<td>a != b</td>
<td>convertible to bool</td>
<td>Equivalent to !(a == b)</td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td>a.swap(b)</td>
<td>void</td>
<td>Effects: exchanges the contents of a and b</td>
<td></td>
<td>(Note A)</td>
</tr>
<tr>
<td>swap(a, b)</td>
<td>void</td>
<td>Equivalent to a.swap(b)</td>
<td></td>
<td>(Note A)</td>
</tr>
<tr>
<td>r = a</td>
<td>X&amp;</td>
<td>Postconditions: r == a.</td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td>a.size()</td>
<td>size_type</td>
<td>distance(a.begin(), a.end())</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.max_size()</td>
<td>size_type</td>
<td>distance(begin(), end()) for the largest possible container</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.empty()</td>
<td>convertible to bool</td>
<td>a.begin() == a.end()</td>
<td></td>
<td>constant</td>
</tr>
</tbody>
</table>

Those entries marked “(Note A)” or “(Note B)” have linear complexity for array and have constant complexity for all other standard containers.

[Note 2: The algorithm equal is defined in Clause 25. — end note]

5 The member function size() returns the number of elements in the container. The number of elements is defined by the rules of constructors, inserts, and erases.

6 begin() returns an iterator referring to the first element in the container. end() returns an iterator which is the past-the-end value for the container. If the container is empty, then begin() == end().

7 In the expressions

i == j
i != j
i < j
i <= j
i >= j
i > j
i <=> j
i - j

where i and j denote objects of a container’s iterator type, either or both may be replaced by an object of the container’s const_iterator type referring to the same element with no change in semantics.

8 Unless otherwise specified, all containers defined in this Clause obtain memory using an allocator (see 16.4.4.6).
Copy constructors for these container types obtain an allocator by calling `allocator_traits<allocator_type>::select_on_container_copy_construction` on the allocator belonging to the container being copied. Move constructors obtain an allocator by move construction from the allocator belonging to the container being moved. Such move construction of the allocator shall not exit via an exception. All other constructors for these container types take a `const allocator_type&` argument.

A copy of this allocator is used for any memory allocation and element construction performed, by these constructors and by all member functions, during the lifetime of each container object or until the allocator is replaced. The allocator may be replaced only via assignment or `swap()`. Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if

\[ \text{allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value,} \]
\[ \text{allocator_traits<allocator_type>::propagate_on_container_move_assignment::value,} \]
\[ \text{allocator_traits<allocator_type>::propagate_on_container_swap::value} \]

is `true` within the implementation of the corresponding container operation. In all container types defined in this Clause, the member `get_allocator()` returns a copy of the allocator used to construct the container or, if that allocator has been replaced, a copy of the most recent replacement.

The expression `a.swap(b)`, for containers `a` and `b` of a standard container type other than `array`, shall exchange the values of `a` and `b` without invoking any move, copy, or swap operations on the individual container elements. Lvalues of any `Compare`, `Pred`, or `Hash` types belonging to `a` and `b` shall be swappable and shall be exchanged by calling `swap` as described in 16.4.4.3. If `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`, then lvalues of type `allocator_type` shall be swappable and the allocators of `a` and `b` shall also be exchanged by calling `swap` as described in 16.4.4.3. Otherwise, the allocators shall not be swapped, and the behavior is undefined unless `a.get_allocator() == b.get_allocator()`. Every iterator referring to an element in one container before the swap shall refer to the same element in the other container after the swap. It is unspecified whether an iterator with value `a.end()` before the swap will have value `b.end()` after the swap.

If the iterator type of a container belongs to the bidirectional or random access iterator categories (23.3), the container is called reversible and meets the additional requirements in Table 74.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::reverse_iterator</code></td>
<td>iterator type whose value type is <code>T</code></td>
<td><code>reverse_iterator&lt;iterator&gt;</code></td>
<td>compile time</td>
</tr>
<tr>
<td><code>X::const_reverse_iterator</code></td>
<td>constant iterator type whose value type is <code>T</code></td>
<td><code>reverse_iterator&lt;const_iterator&gt;</code></td>
<td>compile time</td>
</tr>
<tr>
<td><code>a.rbegin()</code></td>
<td><code>reverse_iterator;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.rend()</code></td>
<td><code>reverse_iterator;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.crbegin()</code></td>
<td><code>const_reverse_iterator</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.crend()</code></td>
<td><code>const_reverse_iterator</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 74: Reversible container requirements

Unless otherwise specified (see 22.2.6.2, 22.2.7.2, 22.3.8.4, and 22.3.11.5) all container types defined in this Clause meet the following additional requirements:
if an exception is thrown by an `insert()` or `emplace()` function while inserting a single element, that function has no effects.

- if an exception is thrown by a `push_back()`, `push_front()`, `emplace_back()`, or `emplace_front()` function, that function has no effects.

- no `erase()`, `clear()`, `pop_back()` or `pop_front()` function throws an exception.

- no copy constructor or assignment operator of a returned iterator throws an exception.

- no `swap()` function throws an exception.

- no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped.

(11.1) Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container.

13 A contiguous container is a container whose member types `iterator` and `const_iterator` meet the `Cpp17RandomAccessIterator` requirements (23.3.5.7) and model `contiguous_iterator` (23.3.4.14).

14 Table 75 lists operations that are provided for some types of containers but not others. Those containers for which the listed operations are provided shall implement the semantics described in Table 75 unless otherwise stated. If the iterators passed to `lexicographical_compare_three_way` meet the `constexpr` iterator requirements (23.3.1) then the operations described in Table 75 are implemented by `constexpr` functions.

Table 75: Optional container operations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/Note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a &lt;=&gt; b</code></td>
<td><code>synth-three-way-result</code></td>
<td><code>lexicographical_compare_three_way(a.begin(), a.end(), b.begin(), b.end(), synth-three-way)</code></td>
<td>Preconditions: Either <code>&lt;&gt;</code> is defined for values of type (possibly const) <code>T</code>, or <code>&lt;</code> is defined for values of type (possibly const) <code>T</code> and <code>&lt;</code> is a total ordering relationship.</td>
<td>linear</td>
</tr>
<tr>
<td><code>&lt;value_type&gt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Note 6: The algorithm `lexicographical_compare_three_way` is defined in Clause 25. — end note]

15 All of the containers defined in this Clause and in 21.3.3 except `array` meet the additional requirements of an allocator-aware container, as described in Table 76.

16 Given an allocator type `A` and given a container type `X` having a `value_type` identical to `T` and an `allocator_traits<type>` identical to `allocator_traits<A>::rebind_alloc<T>` and given an `value m` of type `A`, a pointer `p` of type `T*`, an expression `v` of type (possibly `const`) `T`, and an `rvalue rv` of type `T`, the following terms are defined. If `X` is not allocator-aware, the terms below are defined as if `A` were `allocator<T>` — no allocator object needs to be created and user specializations of `allocator<T>` are not instantiated:

(16.1) — `T` is `Cpp17DefaultInsertable into X` means that the following expression is well-formed:

```
allocator_traits<A>::construct(m, p)
```

(16.2) — An element of `X` is `default-inserted` if it is initialized by evaluation of the expression

```
allocator_traits<A>::construct(m, p)
```

where `p` is the address of the uninitialized storage for the element allocated within `X`.

(16.3) — `T` is `Cpp17MoveInsertable into X` means that the following expression is well-formed:

```
allocator_traits<A>::construct(m, p, rv)
```

and its evaluation causes the following postcondition to hold: The value of `*p` is equivalent to the value of `rv` before the evaluation.
[Note 7: \( \text{rv} \) remains a valid object. Its state is unspecified — end note]

(16.4)  
- \( T \) is \textit{Cpp17CopyInsertable into} \( X \) means that, in addition to \( T \) being \textit{Cpp17MoveInsertable} into \( X \), the following expression is well-formed:

\[
\text{allocator_traits}<A>::\text{construct}(m, p, v)
\]

and its evaluation causes the following postcondition to hold: The value of \( v \) is unchanged and is equivalent to \( *p \).

(16.5)  
- \( T \) is \textit{Cpp17EmplaceConstructible into} \( X \) from \( \text{args} \), for zero or more arguments \( \text{args} \), means that the following expression is well-formed:

\[
\text{allocator_traits}<A>::\text{construct}(m, p, \text{args})
\]

(16.6)  
- \( T \) is \textit{Cpp17Erasable from} \( X \) means that the following expression is well-formed:

\[
\text{allocator_traits}<A>::\text{destroy}(m, p)
\]

[Note 8: A container calls \text{allocator_traits}<A>::\text{construct}(m, p, \text{args}) to construct an element at \( p \) using \text{args}, with \( m == \text{get_allocator()} \). The default \text{construct} in \text{allocator} will call ::new((void*)p) \( T(\text{args}) \), but specialized allocators can choose a different definition. — end note]

In **Table 76**, \( X \) denotes an allocator-aware container class with a value type of \( T \) using allocator of type \( A \). \( u \) denotes a variable, \( a \) and \( b \) denote non-const lvalues of type \( X \), \( t \) denotes an lvalue or a const rvalue of type \( X \), \( rv \) denotes a non-const rvalue of type \( X \), and \( m \) is a value of type \( A \).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{allocator}_- \quad \text{A}</td>
<td>\text{Mandates:}</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>\text{allocator}_- \quad \text{A}</td>
<td>\text{allocator_type::value_type is the same as} ( X::\text{value_type} ).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{get}_- \quad \text{A}</td>
<td>\text{get_allocator()}</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>\text{X()} \quad \text{X u;}</td>
<td>\text{Preconditions:} ( A ) meets the \text{Cpp17DefaultConstructible} requirements.</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>\text{X(m)} \quad \text{X u(m);}</td>
<td>\text{Postconditions:} ( u.\text{empty()} ) \text{returns} \text{true}, ( u.\text{get_allocator()} == A() )</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>\text{X(t, m)} \quad \text{X u(t, m);}</td>
<td>\text{Preconditions:} ( T ) is \textit{Cpp17CopyInsertable} into ( X ).</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>\text{X(rv)} \quad \text{X u(rv);}</td>
<td>\text{Postconditions:} ( u ) has the same elements as ( rv ) had before this construction; the value of ( u.\text{get_allocator()} ) is the same as the value of ( rv.\text{get_allocator()} ) before this construction.</td>
<td>constant if ( m == rv.\text{-}\text{get_allocator()} ), otherwise linear</td>
<td></td>
</tr>
<tr>
<td>\text{X(rv, m)} \quad \text{X u(rv, m);}</td>
<td>\text{Preconditions:} ( T ) is \textit{Cpp17MoveInsertable} into ( X ).</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>\text{X(rv, m)} \quad \text{X u(rv, m);}</td>
<td>\text{Postconditions:} ( u ) has the same elements, or copies of the elements, that ( rv ) had before this construction, ( u.\text{get_allocator()} == m )</td>
<td>linear</td>
<td></td>
</tr>
</tbody>
</table>
### Table 76: Allocator-aware container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = t</code></td>
<td><code>X&amp;</code></td>
<td>Preconditions: <code>T</code> is <code>Cpp17CopyInsertable</code> into <code>X</code> and <code>Cpp17CopyAssignable</code>. Postconditions: <code>a == t</code></td>
<td>linear</td>
</tr>
<tr>
<td><code>a = rv</code></td>
<td><code>X&amp;</code></td>
<td>Preconditions: If <code>allocator_traits&lt;allocator_type&gt;::propagate_on_container_move_assignment::value</code> is <code>false</code>, <code>T</code> is <code>Cpp17MoveInsertable</code> into <code>X</code> and <code>Cpp17MoveAssignable</code>. Effects: All existing elements of <code>a</code> are either move assigned to or destroyed. Postconditions: <code>a</code> is equal to the value that <code>rv</code> had before this assignment.</td>
<td>linear</td>
</tr>
<tr>
<td><code>a.swap(b)</code></td>
<td><code>void</code></td>
<td>Effects: exchanges the contents of <code>a</code> and <code>b</code></td>
<td>constant</td>
</tr>
</tbody>
</table>

---

18 The behavior of certain container member functions and deduction guides depends on whether types qualify as input iterators or allocators. The extent to which an implementation determines that a type cannot be an input iterator is unspecified, except that as a minimum integral types shall not qualify as input iterators. Likewise, the extent to which an implementation determines that a type cannot be an allocator is unspecified, except that as a minimum a type `A` shall not qualify as an allocator unless it meets both of the following conditions:

- The qualified-id `A::value_type` is valid and denotes a type (13.10.3).
- The expression `declval<A&>().allocate(size_t{})` is well-formed when treated as an unevaluated operand.

22.2.2 Container data races

For purposes of avoiding data races (16.4.6.10), implementations shall consider the following functions to be const: `begin`, `end`, `rbegin`, `rend`, `front`, `back`, `data`, `find`, `lower_bound`, `upper_bound`, `equal_range`, `at` and, except in associative or unordered associative containers, `operator[]`.

Notwithstanding 16.4.6.10, implementations are required to avoid data races when the contents of the contained object in different elements in the same container, excepting `vector<bool>`, are modified concurrently.

[Note 1: For a `vector<int> x` with a size greater than one, `x[1] = 5` and `*x.begin() = 10` can be executed concurrently without a data race, but `x[0] = 5` and `*x.begin() = 10` executed concurrently can result in a data race. As an exception to the general rule, for a `vector<bool> y`, `y[0] = true` can race with `y[1] = true`. — end note]

22.2.3 Sequence containers

A sequence container organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides four basic kinds of sequence containers: `vector`, `forward_list`, `list`, and `deque`. In addition, `array` is provided as a sequence container which provides limited sequence operations because it has a fixed number of elements. The library also provides container adaptors that make it easy to construct abstract data types, such as `stacks` or `queues`, out of the basic sequence container kinds (or out of other kinds of sequence containers that the user defines).

[Note 1: The sequence containers offer the programmer different complexity trade-offs. `vector` is appropriate in most circumstances. `array` has a fixed size known during translation. `list` or `forward_list` support frequent insertions and deletions from the middle of the sequence. `deque` supports efficient insertions and deletions taking place at the beginning or at the end of the sequence. When choosing a container, remember `vector` is best; leave a comment to explain if you choose from the rest! — end note]
In Tables 77 and 78, X denotes a sequence container class, a denotes a value of type X containing elements of type T, u denotes the name of a variable being declared, A denotes X::allocator_type if the qualified-id X::allocator_type is valid and denotes a type (13.10.3) and allocator<T> if it doesn’t, i and j denote iterators that meet the Cpp17InputIterator requirements and refer to elements implicitly convertible to value_type, [i, j) denotes a valid range, il designates an object of type initializer_list<value_type>, n denotes a value of type X::size_type, p denotes a valid constant iterator to a, q denotes a valid dereferenceable constant iterator to a, [q1, q2) denotes a valid range of constant iterators in a, t denotes an lvalue or a const rvalue of X::value_type, and rv denotes a non-const rvalue of X::value_type. Args denotes a template parameter pack; args denotes a function parameter pack with the pattern Args&&.

The complexities of the expressions are sequence dependent.

Table 77: Sequence container requirements (in addition to container)  

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
</table>
| X(n, t)    |             | Preconditions: T is Cpp17CopyInsertable into X.  
| X u(n, t); |             | Postconditions: distance(begin(), end()) == n  
|            |             | Effects: Constructs a sequence container with n copies of t |
| X(i, j)    |             | Preconditions: T is Cpp17EmplaceConstructible into X from *i.  
| X u(i, j); |             | For vector, if the iterator does not meet the Cpp17ForwardIterator requirements (23.3.5.5), T is also Cpp17MoveInsertable into X.  
|            |             | Postconditions: distance(begin(), end()) == distance(i, j)  
|            |             | Effects: Constructs a sequence container equal to the range [i, j]. Each iterator in the range [i, j) is dereferenceable exactly once. |
| X(il)      |             | Equivalent to X(il.begin(), il.end())  
| a = il     | X&          | Preconditions: T is Cpp17CopyInsertable into X and Cpp17CopyAssignable.  
|            |             | Effects: Assigns the range [il.begin(), il.end()) into a. All existing elements of a are either assigned to or destroyed.  
|            |             | Returns: *this. |
| aemplace(p, args) | iterator | Preconditions: T is Cpp17EmplaceConstructible into X from args.  
|            |             | For vector and deque, T is also Cpp17MoveInsertable into X and Cpp17MoveAssignable.  
|            |             | Effects: Inserts an object of type T constructed with std::forward<Args>(args)... before p.  
|            |             | [Note 2: args can directly or indirectly refer to a value in a. —end note] |
| a.insert(p, t) | iterator | Preconditions: T is Cpp17CopyInsertable into X. For vector and deque, T is also Cpp17CopyAssignable.  
|            |             | Effects: Inserts a copy of t before p. |
| a.insert(p, rv) | iterator | Preconditions: T is Cpp17MoveInsertable into X. For vector and deque, T is also Cpp17MoveAssignable.  
|            |             | Effects: Inserts a copy of rv before p. |
Table 77: Sequence container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.insert(p, n, t)</code></td>
<td>iterator</td>
<td>Preconditions: T is <code>Cpp17CopyInsertable</code> into X and <code>Cpp17CopyAssignable</code>. Effects: Inserts n copies of t before p.</td>
</tr>
<tr>
<td><code>a.insert(p, i, j)</code></td>
<td>iterator</td>
<td>Preconditions: T is <code>Cpp17EmplaceConstructible</code> into X from *i. For vector and deque, T is also <code>Cpp17MoveInsertable</code> into X, <code>Cpp17MoveConstructible</code>, <code>Cpp17MoveAssignable</code>, and swappable (16.4.4.3). Neither i nor j are iterators into a. Effects: Inserts copies of elements in [i, j) before p. Each iterator in the range [i, j) shall be dereferenced exactly once.</td>
</tr>
<tr>
<td><code>a.insert(p, il)</code></td>
<td>iterator</td>
<td>a.insert(p, il.begin(), il.end()).</td>
</tr>
<tr>
<td><code>a.erase(q)</code></td>
<td>iterator</td>
<td>Preconditions: For vector and deque, T is <code>Cpp17MoveAssignable</code>. Effects: Erases the element pointed to by q.</td>
</tr>
<tr>
<td><code>a.erase(q1, q2)</code></td>
<td>iterator</td>
<td>Preconditions: For vector and deque, T is <code>Cpp17MoveAssignable</code>. Effects: Erases the elements in the range [q1, q2).</td>
</tr>
<tr>
<td><code>a.clear()</code></td>
<td>void</td>
<td>Effects: Destroys all elements in a. Invalidates all references, pointers, and iterators referring to the elements of a and may invalidate the past-the-end iterator. Postconditions: a.empty() is true. Complexity: Linear.</td>
</tr>
<tr>
<td><code>a.assign(i, j)</code></td>
<td>void</td>
<td>Preconditions: T is <code>Cpp17EmplaceConstructible</code> into X from *i and assignable from *i. For vector, if the iterator does not meet the forward iterator requirements (23.3.5.5), T is also <code>Cpp17MoveInsertable</code> into X. Neither i nor j are iterators into a. Effects: Replaces elements in a with a copy of [i, j). Invalidates all references, pointers and iterators referring to the elements of a. For vector and deque, also invalidates the past-the-end iterator. Each iterator in the range [i, j) shall be dereferenced exactly once.</td>
</tr>
<tr>
<td><code>a.assign(il)</code></td>
<td>void</td>
<td>a.assign(il.begin(), il.end()).</td>
</tr>
<tr>
<td><code>a.assign(n, t)</code></td>
<td>void</td>
<td>Preconditions: T is <code>Cpp17CopyInsertable</code> into X and <code>Cpp17CopyAssignable</code>. t is not a reference into a. Effects: Replaces elements in a with n copies of t. Invalidates all references, pointers and iterators referring to the elements of a. For vector and deque, also invalidates the past-the-end iterator.</td>
</tr>
</tbody>
</table>

5 The iterator returned from `a.insert(p, t)` points to the copy of t inserted into a.

6 The iterator returned from `a.insert(p, rv)` points to the copy of rv inserted into a.
The iterator returned from `a.insert(p, n, t)` points to the copy of the first element inserted into `a`, or `p` if `n == 0`.

The iterator returned from `a.insert(p, i, j)` points to the copy of the first element inserted into `a`, or `p` if `i == j`.

The iterator returned from `a.insert(p, il)` points to the copy of the first element inserted into `a`, or `p` if `il` is empty.

The iterator returned from `a.emplace(p, args)` points to the new element constructed from `args` into `a`.

The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased. If no such element exists, `a.end()` is returned.

The iterator returned by `a.erase(q1, q2)` points to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

For every sequence container defined in this Clause and in Clause 21:

13.1 — If the constructor

```
template<class InputIterator>
X(InputIterator first, InputIterator last,
const allocator_type& alloc = allocator_type());
```

is called with a type `InputIterator` that does not qualify as an input iterator, then the constructor shall not participate in overload resolution.

13.2 — If the member functions of the forms:

```
template<class InputIterator>
return-type F(const_iterator p,
InputIterator first, InputIterator last); // such as insert
```

```
template<class InputIterator>
return-type F(InputIterator first, InputIterator last); // such as append, assign
```

```
template<class InputIterator>
return-type F(const_iterator i1, const_iterator i2,
InputIterator first, InputIterator last); // such as replace
```

are called with a type `InputIterator` that does not qualify as an input iterator, then these functions shall not participate in overload resolution.

13.3 — A deduction guide for a sequence container shall not participate in overload resolution if it has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter, or if it has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.

Table 78 lists operations that are provided for some types of sequence containers but not others. An implementation shall provide these operations for all container types shown in the “container” column, and shall implement them so as to take amortized constant time.

### Table 78: Optional sequence container operations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.front()</code></td>
<td>reference; const_reference</td>
<td><code>*a.begin()</code></td>
<td>basic_string, array, deque, forward_list, list, vector</td>
</tr>
</tbody>
</table>
| `a.back()` | reference; const_reference | `{ auto tmp = a.end();
--tmp;
return *tmp; }` | basic_string, array, deque, list, vector |
Table 78: Optional sequence container operations (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.emplace_front(args)</td>
<td>reference</td>
<td>Effects: Prepends an object of type T constructed with std::forward&lt;Args&gt;(args)...</td>
<td>deque,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: T is Cpp17EmplaceConstructible into X from args.</td>
<td>forward_list,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns: a.front().</td>
<td>list</td>
</tr>
<tr>
<td>a.emplace_back(args)</td>
<td>reference</td>
<td>Effects: Appends an object of type T constructed with std::forward&lt;Args&gt;(args)...</td>
<td>deque,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: T is Cpp17EmplaceConstructible into X from args. For vector, T is also Cpp17MoveInsertable into X.</td>
<td>list, vector</td>
</tr>
<tr>
<td>a.push_front(t)</td>
<td>void</td>
<td>Effects: Prepends a copy of t.</td>
<td>deque,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: T is Cpp17CopyInsertable into X.</td>
<td>forward_list,</td>
</tr>
<tr>
<td>a.push_front(rv)</td>
<td>void</td>
<td>Effects: Prepends a copy of rv.</td>
<td>deque,</td>
</tr>
<tr>
<td>a.push_back(t)</td>
<td>void</td>
<td>Effects: Appends a copy of t.</td>
<td>basic_string,</td>
</tr>
<tr>
<td>a.push_back(rv)</td>
<td>void</td>
<td>Effects: Appends a copy of rv.</td>
<td>basic_string,</td>
</tr>
<tr>
<td>a.pop_front()</td>
<td>void</td>
<td>Effects: Destroys the first element.</td>
<td>deque,</td>
</tr>
<tr>
<td>a.pop_back()</td>
<td>void</td>
<td>Effects: Destroys the last element.</td>
<td>basic_string,</td>
</tr>
<tr>
<td>[n]</td>
<td>reference;</td>
<td>*(a.begin() + n)</td>
<td>basic_string,</td>
</tr>
<tr>
<td>a[n]</td>
<td>const_reference</td>
<td>for constant a</td>
<td>array, deque,</td>
</tr>
<tr>
<td>a.at(n)</td>
<td>reference;</td>
<td>*(a.begin() + n)</td>
<td>basic_string,</td>
</tr>
<tr>
<td></td>
<td>const_reference</td>
<td>for constant a</td>
<td>array, deque,</td>
</tr>
</tbody>
</table>

15 The member function at() provides bounds-checked access to container elements. at() throws out_of_range if n >= a.size().

22.2.4 Node handles

22.2.4.1 Overview

A node handle is an object that accepts ownership of a single element from an associative container (22.2.6) or an unordered associative container (22.2.7). It may be used to transfer that ownership to another container with compatible nodes. Containers with compatible nodes have the same node handle type. Elements may be transferred in either direction between container types in the same row of Table 79.

1 If a node handle is not empty, then it contains an allocator that is equal to the allocator of the container when the element was extracted. If a node handle is empty, it contains no allocator.
Table 79: Container types with compatible nodes  [tab:container.node.compat]

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Container Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>map&lt;K, T, C1, A&gt;</td>
<td>map&lt;K, T, C2, A&gt;</td>
</tr>
<tr>
<td>map&lt;K, T, C1, A&gt;</td>
<td>multimap&lt;K, T, C2, A&gt;</td>
</tr>
<tr>
<td>set&lt;K, C1, A&gt;</td>
<td>set&lt;K, C2, A&gt;</td>
</tr>
<tr>
<td>set&lt;K, C1, A&gt;</td>
<td>multiset&lt;K, C2, A&gt;</td>
</tr>
<tr>
<td>unordered_map&lt;K, T, H1, E1, A&gt;</td>
<td>unordered_map&lt;K, T, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_map&lt;K, T, H1, E1, A&gt;</td>
<td>unordered_multimap&lt;K, T, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_set&lt;K, H1, E1, A&gt;</td>
<td>unordered_set&lt;K, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_set&lt;K, H1, E1, A&gt;</td>
<td>unordered_multiset&lt;K, H2, E2, A&gt;</td>
</tr>
</tbody>
</table>

3 Class node-handle is for exposition only.

4 If a user-defined specialization of pair exists for pair<\text{const} \text{Key}, \text{T}> or pair<\text{Key}, \text{T}>, where \text{Key} is the container’s key_type and \text{T} is the container’s mapped_type, the behavior of operations involving node handles is undefined.

template<\text{unspe}cified>
class node-handle {
public:
  // These type declarations are described in Tables 80 and 81.
  using value_type = see below;  // not present for map containers
  using key_type = see below;    // not present for set containers
  using mapped_type = see below; // not present for set containers
  using allocator_type = see below;

private:
  using container_node_type = unspecified;
  using ator_traits = allocator_traits<allocator_type>;

  typename ator_traits::template rebind_traits<container_node_type>::pointer ptr_;
  optional<allocator_type> alloc_;

public:
  // 22.2.4.2, constructors, copy, and assignment
  constexpr node-handle() noexcept : ptr_(), alloc_() {}
  node-handle(node-handle&&) noexcept;
  node-handle& operator=(node-handle&&);

  // 22.2.4.3, destructor
  ~node-handle();

  // 22.2.4.4, observers
  value_type& value() const;  // not present for map containers
  key_type& key() const;      // not present for set containers
  mapped_type& mapped() const; // not present for set containers

  allocator_type get_allocator() const;
  explicit operator bool() const noexcept;
  [[nodiscard]] bool empty() const noexcept;

  // 22.2.4.5, modifiers
  void swap(node-handle&) noexcept(ator_traits::propagate_on_container_swap::value ||
    ator_traits::is_always_equal::value);

  friend void swap(node-handle& x, node-handle& y) noexcept(noexcept(x.swap(y))) {
    x.swap(y);
  }
};
22.2.4.2 Constructors, copy, and assignment

node-handle(node-handle& nh) noexcept;

Effects: Constructs a node-handle object initializing ptr_ with nh.ptr_. Move constructs alloc_ with nh.alloc_. Assigns nullptr to nh.ptr_ and assigns nullopt to nh.alloc_.

node-handle& operator=(node-handle&& nh);

Preconditions: Either !alloc_, or ator_traits::propagate_on_container_move_assignment::value is true, or alloc_ == nh.alloc_.

Effects:
(3.1) If ptr_ != nullptr, destroys the value_type subobject in the container_node_type object pointed to by ptr_ by calling ator_traits::destroy, then deallocates ptr_ by calling ator_traits::template rebind_traits<container_node_type>::deallocate.
(3.2) Assigns nh.ptr_ to ptr_.
(3.3) If !alloc_ or ator_traits::propagate_on_container_move_assignment::value is true, move assigns nh.alloc_ to alloc_.
(3.4) Assigns nullptr to nh.ptr_ and assigns nullopt to nh.alloc_.

Returns: *this.

Throws: Nothing.

22.2.4.3 Destructor

~node-handle();

Effects: If ptr_ != nullptr, destroys the value_type subobject in the container_node_type object pointed to by ptr_ by calling ator_traits::destroy, then deallocates ptr_ by calling ator_traits::template rebind_traits<container_node_type>::deallocate.

22.2.4.4 Observers

value_type& value() const;

Preconditions: empty() == false.

Returns: A reference to the value_type subobject in the container_node_type object pointed to by ptr_.

Throws: Nothing.

key_type& key() const;

Preconditions: empty() == false.

Returns: A non-const reference to the key_type member of the value_type subobject in the container_node_type object pointed to by ptr_.

Throws: Nothing.

Remarks: Modifying the key through the returned reference is permitted.

mapped_type& mapped() const;

Preconditions: empty() == false.

Returns: A reference to the mapped_type member of the value_type subobject in the container_node_type object pointed to by ptr_.

Throws: Nothing.

allocator_type get_allocator() const;

Preconditions: empty() == false.

Returns: *alloc_.

Throws: Nothing.
explicit operator bool() const noexcept;

**Returns**: ptr_ != nullptr.

[[nodiscard]] bool empty() const noexcept;

**Returns**: ptr_ == nullptr.

### 22.2.4.5 Modifiers

```cpp
void swap(node-handle& nh)
    noexcept(ator_traits::propagate_on_container_swap::value ||
             ator_traits::is_always_equal::value);
```

**Preconditions**: !alloc_, or !nh.alloc_, or ator_traits::propagate_on_container_swap::value is true, or alloc_ == nh.alloc_.

**Effects**: Calls swap(ptr_, nh.ptr_). If !alloc_, or !nh.alloc_, or ator_traits::propagate_on_container_swap::value is true calls swap(alloc_, nh.alloc_).

### 22.2.5 Insert return type

The associative containers with unique keys and the unordered containers with unique keys have a member function `insert` that returns a nested type `insert_return_type`. That return type is a specialization of the template specified in this subclause.

```cpp
template<class Iterator, class NodeType>
struct insert-return-type
{
    Iterator position;
    bool inserted;
    NodeType node;
};
```

The name `insert-return-type` is exposition only. `insert-return-type` has the template parameters, data members, and special members specified above. It has no base classes or members other than those specified.

### 22.2.6 Associative containers

#### 22.2.6.1 General

Associtative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: `set`, `multiset`, `map` and `multimap`.

Each associative container is parameterized on `Key` and an ordering relation `Compare` that induces a strict weak ordering (25.8) on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary `mapped type` `T` with the `Key`. The object of type `Compare` is called the `comparison object` of a container.

The phrase “equivalence of keys” means the equivalence relation imposed by the comparison object. That is, two keys `k1` and `k2` are considered to be equivalent if for the comparison object `comp`, `comp(k1, k2) == false` & `comp(k2, k1) == false`.

**Note 1**: This is not necessarily the same as the result of `k1 == k2`. —end note

For any two keys `k1` and `k2` in the same container, calling `comp(k1, k2)` shall always return the same value.

An associative container supports `unique keys` if it may contain at most one element for each key. Otherwise, it supports `equivalent keys`. The `set` and `map` classes support unique keys; the `multiset` and `multimap` classes support equivalent keys. For `multiset` and `multimap`, `insert`, `emplace`, and `erase` preserve the relative ordering of equivalent elements.

For `set` and `multiset` the value type is the same as the key type. For `map` and `multimap` it is equal to `pair<Const Key, T>`.

`iterator` of an associative container is of the bidirectional iterator category. For associative containers where the value type is the same as the key type, both `iterator` and `const_iterator` are constant iterators. It is unspecified whether or not `iterator` and `const_iterator` are the same type.

**Note 2**: `iterator` and `const_iterator` have identical semantics in this case, and `iterator` is convertible to `const_iterator`. Users can avoid violating the one-definition rule by always using `const_iterator` in their function parameter lists. —end note
The associative containers meet all the requirements of Allocator-aware containers (22.2.1), except that for map and multimap, the requirements placed on value_type in Table 76 apply instead to key_type and mapped_type.

[Note 3: For example, in some cases key_type and mapped_type are required to be Cpp17CopyAssignable even though the associated value_type, pair<const key_type, mapped_type>, is not Cpp17CopyAssignable. — end note]

8 In Table 80, X denotes an associative container class, a denotes a value of type X, a2 denotes a value of a type with nodes compatible with type X (Table 79), b denotes a possibly const value of type X, u denotes the name of a variable being declared, a_uniq denotes a value of type X when X supports unique keys, a_eq denotes a value of type X when X supports multiple keys, a_tran denotes a possibly const value of type X when the qualified-id X::key_compare::is_transparent is valid and denotes a type (13.10.3), i and j meet the Cpp17InputIterator requirements and refer to elements implicitly convertible to value_type, [i, j) denotes a valid range, p denotes a valid constant iterator to a, q denotes a valid dereferenceable constant iterator to a, r denotes a valid dereferenceable iterator to a, [q1, q2) denotes a valid range of constant iterators in a, il designates an object of type initializer_list<value_type>, t denotes a value of type X::value_type, k denotes a value of type X::key_type and c denotes a possibly const value of type X::key_compare; kl is a value such that a is partitioned (25.8) with respect to c(r, kl), with r the key value of e and e in a; ku is a value such that a is partitioned with respect to !c(ku, r); ke is a value such that a is partitioned with respect to c(r, ke) and !c(ke, r), with c(r, ke) implying !c(ke, r). A denotes the storage allocator used by X, if any, or allocator<X::value_type> otherwise, m denotes an allocator of a type convertible to A, and nh denotes a non-const rvalue of type X::node_type.

Table 80: Associative container requirements (in addition to container)  [tab:container.assoc.req]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::key_type</td>
<td>Key</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::mapped_-type (map and multimap only)</td>
<td>T</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::value_-type (set and multiset only)</td>
<td>Key</td>
<td>Preconditions: value_type is Cpp17Erasable from X</td>
<td>compile time</td>
</tr>
<tr>
<td>X::value_-type (map and multimap only)</td>
<td>pair&lt;const Key, T&gt;</td>
<td>Preconditions: value_type is Cpp17Erasable from X</td>
<td>compile time</td>
</tr>
<tr>
<td>X::key_-compare</td>
<td>Compare</td>
<td>Preconditions: key_compare is Cpp17CopyConstructible.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::value_-compare</td>
<td>a binary predicate type</td>
<td>is the same as key_compare for set and multiset; is an ordering relation on pairs induced by the first component (i.e., Key) for map and multimap</td>
<td>compile time</td>
</tr>
<tr>
<td>X::node_-type</td>
<td>A specialization of a node-handle class template, such that the public nested types are the same types as the corresponding types in X.</td>
<td>see 22.2.4</td>
<td>compile time</td>
</tr>
</tbody>
</table>
Table 80: Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X(c)</code></td>
<td><code>X u(c);</code></td>
<td><strong>Effects:</strong> Constructs an empty container. Uses a copy of <code>c</code> as a comparison object.</td>
<td>constant</td>
</tr>
<tr>
<td><code>X()</code></td>
<td><code>X u;</code></td>
<td><strong>Preconditions:</strong> <code>key_compare</code> meets the <code>Cpp17DefaultConstructible</code> requirements. <strong>Effects:</strong> Constructs an empty container. Uses <code>Compare()</code> as a comparison object.</td>
<td>constant</td>
</tr>
<tr>
<td><code>X(i,j,c)</code></td>
<td><code>X u(i,j,c);</code></td>
<td><strong>Preconditions:</strong> <code>value_type</code> is <code>Cpp17EmplaceConstructible</code> into <code>X</code> from <code>*i</code>. <strong>Effects:</strong> Constructs an empty container and inserts elements from the range <code>[i, j)</code> into it; uses <code>c</code> as a comparison object.</td>
<td><code>N \log N</code> in general, where <code>N</code> has the value <code>distance(i, j)</code>; linear if <code>[i, j)</code> is sorted with <code>value_comp()</code></td>
</tr>
<tr>
<td><code>X(i,j)</code></td>
<td><code>X u(i,j);</code></td>
<td><strong>Preconditions:</strong> <code>key_compare</code> meets the <code>Cpp17DefaultConstructible</code> requirements. <code>value_type</code> is <code>Cpp17EmplaceConstructible</code> into <code>X</code> from <code>*i</code>. <strong>Effects:</strong> Same as above, but uses <code>Compare()</code> as a comparison object.</td>
<td>same as above</td>
</tr>
<tr>
<td><code>X(il)</code></td>
<td><code>X u(il);</code></td>
<td>same as <code>X(il.begin(), il.end())</code></td>
<td>same as <code>X(il.begin(), il.end())</code></td>
</tr>
<tr>
<td><code>X(il,c)</code></td>
<td><code>X u(il,c);</code></td>
<td>same as <code>X(il.begin(), il.end(), c)</code></td>
<td>same as <code>X(il.begin(), il.end(), c)</code></td>
</tr>
<tr>
<td><code>a = il</code></td>
<td><code>X &amp;</code></td>
<td><strong>Preconditions:</strong> <code>value_type</code> is <code>Cpp17CopyInsertable</code> into <code>X</code> and <code>Cpp17CopyAssignable</code>. <strong>Effects:</strong> Assigns the range <code>[il.begin(), il.end())</code> into <code>a</code>. All existing elements of <code>a</code> are either assigned to or destroyed.</td>
<td><code>N \log N</code> in general, where <code>N</code> has the value <code>il.size() + a.size();</code> linear if <code>[il.begin(), il.end())</code> is sorted with <code>value_comp()</code></td>
</tr>
<tr>
<td><code>b.key_-comp()</code></td>
<td><code>X: :key_-compare</code></td>
<td><strong>Returns:</strong> the comparison object out of which <code>b</code> was constructed.</td>
<td>constant</td>
</tr>
<tr>
<td><code>b.value_-comp()</code></td>
<td><code>X: :value_-compare</code></td>
<td><strong>Returns:</strong> an object of <code>value_compare</code> constructed out of the comparison object</td>
<td>constant</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Assertion/note</td>
<td>Complexity</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
</tbody>
</table>
| a_uniq. emplace(args) | pair< iterator, bool> | **Preconditions:** value\_type is `Cpp17EmplaceConstructible` into X from args.  
`Effects:` Inserts a value\_type object t constructed with `std::forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of t. | logarithmic |
| a_eq. emplace(args)                    | iterator         | **Preconditions:** value\_type is `Cpp17EmplaceConstructible` into X from args.  
`Effects:` Inserts a value\_type object t constructed with `std::forward<Args>(args)...` and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to t exists in a\_eq, t is inserted at the end of that range. | logarithmic |
| a.emplace\_hint(p, args)               | iterator         | **Effects:** Equivalent to a\_emplace( std::forward<Args>(args)...). Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The element is inserted as close as possible to the position just prior to p. | logarithmic in general, but amortized constant if the element is inserted right before p |
| a_uniq. insert(t)                      | pair< iterator, bool> | **Preconditions:** If t is a non-const rvalue, value\_type is `Cpp17MoveInsertable` into X; otherwise, value\_type is `Cpp17CopyInsertable` into X.  
`Effects:` Inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of t. | logarithmic |
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a_eq. insert(t)</code></td>
<td>iterator</td>
<td><em>Preconditions:</em> If <code>t</code> is a non-const rvalue, <code>value_type</code> is <code>Cpp17MoveInsertable</code> into <code>X</code>; otherwise, <code>value_type</code> is <code>Cpp17CopyInsertable</code> into <code>X</code>. <em>Effects:</em> Inserts <code>t</code> and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to <code>t</code> exists in <code>a_eq</code>, <code>t</code> is inserted at the end of that range.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a.insert(p, t)</code></td>
<td>iterator</td>
<td><em>Preconditions:</em> If <code>t</code> is a non-const rvalue, <code>value_type</code> is <code>Cpp17MoveInsertable</code> into <code>X</code>; otherwise, <code>value_type</code> is <code>Cpp17CopyInsertable</code> into <code>X</code>. <em>Effects:</em> Inserts <code>t</code> if and only if there is no element with key equivalent to the key of <code>t</code> in containers with unique keys; always inserts <code>t</code> in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to the key of <code>t</code>. <code>t</code> is inserted as close as possible to the position just prior to <code>p</code>.</td>
<td>logarithmic in general, but amortized constant if <code>t</code> is inserted right before <code>p</code>.</td>
</tr>
<tr>
<td><code>a.insert(i, j)</code></td>
<td>void</td>
<td><em>Preconditions:</em> <code>value_type</code> is <code>Cpp17EmplaceConstructible</code> into <code>X</code> from <code>*i</code>. Neither <code>i</code> nor <code>j</code> are iterators into <code>a</code>. <em>Effects:</em> Inserts each element from the range <code>[i, j)</code> if and only if there is no element with key equivalent to the key of that element in containers with unique keys; always inserts that element in containers with equivalent keys.</td>
<td>$N \log(a.size() + N)$, where $N$ has the value <code>distance(i, j)</code></td>
</tr>
<tr>
<td><code>a.insert(il)</code></td>
<td>void</td>
<td><em>Effects:</em> Equivalent to <code>a.insert(il.begin(), il.end())</code></td>
<td></td>
</tr>
</tbody>
</table>
Table 80: Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_uniq.</td>
<td>insert_-</td>
<td>Preconditions: nh is empty or a_uniq.get_allocator() == nh.get_allocator(). Effects: If nh is empty, has no effect. Otherwise, inserts the element owned by nh if and only if there is no element in the container with a key equivalent to nh.key(). Postconditions: If nh is empty, inserted is false, position is end(), and node is empty. Otherwise if the insertion took place, inserted is true, position points to the inserted element, and node is empty; if the insertion failed, inserted is false, node has the previous value of nh, and position points to an element with a key equivalent to nh.key().</td>
<td>logarithmic</td>
</tr>
<tr>
<td>insert(nh)</td>
<td>iterator</td>
<td>Preconditions: nh is empty or a_eq.get_allocator() == nh.get_allocator(). Effects: If nh is empty, has no effect and returns a_eq.end(). Otherwise, inserts the element owned by nh and returns an iterator pointing to the newly inserted element. If a range containing elements with keys equivalent to nh.key() exists in a_eq, the element is inserted at the end of that range. Postconditions: nh is empty.</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>

§ 22.2.6.1
Table 80: Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.insert(p, nh)</code></td>
<td>iterator</td>
<td><em>Preconditions:</em> nh is empty or <code>a.get_allocator() == nh.get_allocator()</code>&lt;br&gt;Effects: If nh is empty, has no effect and returns <code>a.end()</code>&lt;br&gt;Otherwise, inserts the element owned by nh if and only if there is no element with key equivalent to <code>nh.key()</code> in containers with unique keys; always inserts the element owned by nh in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to <code>nh.key()</code>. The element is inserted as close as possible to the position just prior to p.&lt;br&gt;<em>Postconditions:</em> nh is empty if insertion succeeds, unchanged if insertion fails.</td>
<td>logarithmic in general, but amortized constant if the element is inserted right before p.</td>
</tr>
<tr>
<td><code>a.extract(k)</code></td>
<td>node_type</td>
<td><em>Effects:</em> Removes the first element in the container with key equivalent to k.&lt;br&gt;<em>Returns:</em> A <code>node_type</code> owning the element if found, otherwise an empty <code>node_type</code>.</td>
<td>log(<code>a.size()</code>)</td>
</tr>
<tr>
<td><code>a.extract(q)</code></td>
<td>node_type</td>
<td><em>Effects:</em> Removes the element pointed to by q.&lt;br&gt;<em>Returns:</em> A <code>node_type</code> owning that element.</td>
<td>amortized constant</td>
</tr>
<tr>
<td><code>a.merge(a2)</code></td>
<td>void</td>
<td><em>Preconditions:</em> <code>a.get_allocator() == a2.get_allocator()</code>&lt;br&gt;<em>Effects:</em> Attempts to extract each element in a2 and insert it into a using the comparison object of a. In containers with unique keys, if there is an element in a with key equivalent to the key of an element from a2, then that element is not extracted from a2.&lt;br&gt;<em>Postconditions:</em> Pointers and references to the transferred elements of a2 refer to those same elements but as members of a. Iterators referring to the transferred elements will continue to refer to their elements, but they now behave as iterators into a, not into a2.&lt;br&gt;<em>Throws:</em> Nothing unless the comparison object throws.</td>
<td>$N \log(a.size() + N)$, where $N$ has the value a2.size().</td>
</tr>
</tbody>
</table>
Table 80: Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.erase(k)</code></td>
<td><code>size_type</code></td>
<td>Effects: Erases all elements in the container with key equivalent to k. Returns: The number of erased elements.</td>
<td>( \log(a.size()) + a.count(k) )</td>
</tr>
<tr>
<td><code>a.erase(q)</code></td>
<td><code>iterator</code></td>
<td>Effects: Erases the element pointed to by q. Returns: An iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns a.end().</td>
<td>amortized constant</td>
</tr>
<tr>
<td><code>a.erase(r)</code></td>
<td><code>iterator</code></td>
<td>Effects: Erases the element pointed to by r. Returns: An iterator pointing to the element immediately following r prior to the element being erased. If no such element exists, returns a.end().</td>
<td>amortized constant</td>
</tr>
<tr>
<td><code>a.erase(q1, q2)</code></td>
<td><code>iterator</code></td>
<td>Effects: Erases all the elements in the range ([q1, q2)). Returns: An iterator pointing to the element pointed to by q2 prior to any elements being erased. If no such element exists, a.end() is returned.</td>
<td>( \log(a.size()) + N ), where ( N ) has the value distance(q1, q2).</td>
</tr>
<tr>
<td><code>a.clear()</code></td>
<td><code>void</code></td>
<td>Effects: Equivalent to <code>a.erase(a.begin(), a.end())</code>. Postconditions: <code>a.empty()</code> is true.</td>
<td>linear in a.size().</td>
</tr>
<tr>
<td><code>b.find(k)</code></td>
<td><code>iterator; const_iterator for constant b.</code></td>
<td>Returns: An iterator pointing to an element with the key equivalent to k, or b.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a_tran.find(ke)</code></td>
<td><code>iterator; const_iterator for constant a_tran.</code></td>
<td>Returns: An iterator pointing to an element with key r such that ( !c(r, ke) &amp;&amp; !c(ke, r) ), or a_tran.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>b.count(k)</code></td>
<td><code>size_type</code></td>
<td>Returns: The number of elements with key equivalent to k.</td>
<td>( \log(b.size()) + b.count(k) )</td>
</tr>
<tr>
<td><code>a_tran.count(ke)</code></td>
<td><code>size_type</code></td>
<td>Returns: The number of elements with key r such that ( !c(r, ke) &amp;&amp; !c(ke, r) ).</td>
<td>( \log(a_tran.size()) + a_tran.count(ke) )</td>
</tr>
<tr>
<td><code>b.contains(k)</code></td>
<td><code>bool</code></td>
<td>Effects: Equivalent to: return <code>b.find(k) != b.end();</code></td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a_tran.contains(ke)</code></td>
<td><code>bool</code></td>
<td>Effects: Equivalent to: return <code>a_tran.find(ke) != a_tran.end();</code></td>
<td>logarithmic</td>
</tr>
</tbody>
</table>
Table 80: Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.lower_bound(k)</td>
<td>iterator; const_iterator for constant b.</td>
<td>Returns: An iterator pointing to the first element with key not less than k, or b.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_tran.lower_bound(kl)</td>
<td>iterator; const_iterator for constant a_tran.</td>
<td>Returns: An iterator pointing to the first element with key such that !c(r, kl), or a_tran.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>b.upper_bound(k)</td>
<td>iterator; const_iterator for constant b.</td>
<td>Returns: An iterator pointing to the first element with key greater than k, or b.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_tran.upper_bound(ku)</td>
<td>iterator; const_iterator for constant a_tran.</td>
<td>Returns: An iterator pointing to the first element with key such that c(ku, r), or a_tran.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>b.equal_range(k)</td>
<td>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for constant b.</td>
<td>Effects: Equivalent to: return make_pair(b.lower_bound(k), b.upper_bound(k));</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_tran.equal_range(ke)</td>
<td>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for constant a_tran.</td>
<td>Effects: Equivalent to: return make_pair(a_tran.lower_bound(ke), a_tran.upper_bound(ke));</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>

9 The **insert** and **emplace** members shall not affect the validity of iterators and references to the container, and the **erase** members shall invalidate only iterators and references to the erased elements.

10 The **extract** members invalidate only iterators to the removed element; pointers and references to the removed element remain valid. However, accessing the element through such pointers and references while the element is owned by a **node_type** is undefined behavior. References and pointers to an element obtained while it is owned by a **node_type** are invalidated if the element is successfully inserted.

11 The fundamental property of iterators of associative containers is that they iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them. For any two dereferenceable iterators i and j such that distance from i to j is positive, the following condition holds:

\[
\text{value_comp}(\ast j, \ast i) = \text{false}
\]

12 For associative containers with unique keys the stronger condition holds:

\[
\text{value_comp}(\ast i, \ast j) \neq \text{false}
\]

13 When an associative container is constructed by passing a comparison object the container shall not store a pointer or reference to the passed object, even if that object is passed by reference. When an associative container is copied, through either a copy constructor or an assignment operator, the target container shall
then use the comparison object from the container being copied, as if that comparison object had been passed to the target container in its constructor.

14 The member function templates `find`, `count`, `contains`, `lower_bound`, `upper_bound`, and `equal_range` shall not participate in overload resolution unless the `qualified-id` `Compare::is_transparent` is valid and denotes a type (13.10.3).

15 A deduction guide for an associative container shall not participate in overload resolution if any of the following are true:

(15.1) — It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.

(15.2) — It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.

(15.3) — It has a `Compare` template parameter and a type that qualifies as an allocator is deduced for that parameter.

22.2.6.2 Exception safety guarantees

[associative.reqmts.except]

1 For associative containers, no `clear()` function throws an exception. `erase(k)` does not throw an exception unless that exception is thrown by the container’s `Compare` object (if any).

2 For associative containers, if an exception is thrown by any operation from within an `insert` or `emplace` function inserting a single element, the insertion has no effect.

3 For associative containers, no `swap` function throws an exception unless that exception is thrown by the swap of the container’s `Compare` object (if any).

22.2.7 Unordered associative containers

[unord.req]

22.2.7.1 General

[unord.req.general]

1 Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four unordered associative containers: `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.

2 Unordered associative containers conform to the requirements for Containers (22.2), except that the expressions `a == b` and `a != b` have different semantics than for the other container types.

3 Each unordered associative container is parameterized by `Key`, by a function object type `Hash` that meets the Cpp17Hash requirements (16.4.4.5) and acts as a hash function for argument values of type `Key`, and by a binary predicate `Pred` that induces an equivalence relation on values of type `Key`. Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary `mapped type` `T` with the `Key`.

4 The container’s object of type `Hash` — denoted by `hash` — is called the `hash function` of the container. The container’s object of type `Pred` — denoted by `pred` — is called the `key equality predicate` of the container.

5 Two values `k1` and `k2` are considered equivalent if the container’s key equality predicate `pred(k1, k2)` is valid and returns `true` when passed those values. If `k1` and `k2` are equivalent, the container’s hash function shall return the same value for both.

[Note 1: Thus, when an unordered associative container is instantiated with a non-default `Pred` parameter it usually needs a non-default `Hash` parameter as well. — end note]

For any two keys `k1` and `k2` in the same container, calling `pred(k1, k2)` shall always return the same value. For any key `k` in a container, calling `hash(k)` shall always return the same value.

6 An unordered associative container supports `unique keys` if it may contain at most one element for each key. Otherwise, it supports `equivalent keys`. `unordered_set` and `unordered_map` support unique keys. `unordered_multiset` and `unordered_multimap` support equivalent keys. In containers that support equivalent keys, elements with equivalent keys are adjacent to each other in the iteration order of the container. Thus, although the absolute order of elements in an unordered container is not specified, its elements are grouped into `equivalent-key groups` such that all elements of each group have equivalent keys. Mutating operations on unordered containers shall preserve the relative order of elements within each equivalent-key group unless otherwise specified.

7 For `unordered_set` and `unordered_multiset` the value type is the same as the key type. For `unordered_map` and `unordered_multimap` it is `pair<const Key, T>`.
For unordered containers where the value type is the same as the key type, both iterator and const_iterator are constant iterators. It is unspecified whether or not iterator and const_iterator are the same type.

[Note 2: iterator and const_iterator have identical semantics in this case, and iterator is convertible to const_iterator. Users can avoid violating the one-definition rule by always using const_iterator in their function parameter lists. —end note]

The elements of an unordered associative container are organized into buckets. Keys with the same hash code appear in the same bucket. The number of buckets is automatically increased as elements are added to an unordered associative container, so that the average number of elements per bucket is kept below a bound. Rehashing invalidates iterators, changes ordering between elements, and changes which buckets elements appear in, but does not invalidate pointers or references to elements. For unordered_multiset and unordered_multimap, rehashing preserves the relative ordering of equivalent elements.

The unordered associative containers meet all the requirements of Allocator-aware containers (22.2.1), except that for unordered_map and unordered_multimap, the requirements placed on value_type in Table 76 apply instead to key_type and mapped_type.

[Note 3: For example, key_type and mapped_type are sometimes required to be Cpp17CopyAssignable even though the associated value_type, pair<const key_type, mapped_type>, is not Cpp17CopyAssignable. —end note]

In Table 81:

- X denotes an unordered associative container class,
- a denotes a value of type X,
- a2 denotes a value of a type with nodes compatible with type X (Table 79),
- b denotes a possibly const value of type X,
- a_uniq denotes a value of type X when X supports unique keys,
- a_eq denotes a value of type X when X supports equivalent keys,
- a_tran denotes a possibly const value of type X when the qualified-ids X::key_equal::is_transparent and X::hasher::is_transparent are both valid and denote types (13.10.3),
- i and j denote input iterators that refer to value_type,
- [i, j) denotes a valid range,
- p and q2 denote valid constant iterators to a,
- q and q1 denote valid dereferenceable constant iterators to a,
- r denotes a valid dereferenceable iterator to a,
- [q1, q2) denotes a valid range in a,
- il denotes a value of type initializer_list<value_type>,
- t denotes a value of type X::value_type,
- k denotes a value of type key_type,
- hf denotes a possibly const value of type hasher,
- eq denotes a possibly const value of type key_equal,
- ke is a value such that
  - eq(r1, ke) == eq(ke, r1)
  - hf(r1) == hf(ke) if eq(r1, ke) is true, and
  - (eq(r1, ke) && eq(r1, r2)) == eq(r2, ke)
  where r1 and r2 are keys of elements in a_tran,
- n denotes a value of type size_type,
- z denotes a value of type float, and
- nh denotes a non-const rvalue of type X::node_type.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::key_type</td>
<td>Key</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::mapped_type</td>
<td>T</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::value_type</td>
<td>Key</td>
<td>Preconditions: value_type is Cpp17Erasable from X</td>
<td>compile time</td>
</tr>
<tr>
<td>X::value_type</td>
<td>pair&lt;const Key, T&gt;</td>
<td>Preconditions: value_type is Cpp17Erasable from X</td>
<td>compile time</td>
</tr>
<tr>
<td>X::hasher</td>
<td>Hash</td>
<td>Preconditions: Hash is a unary function object type such that the expression hf(k) has type size_t.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::key_equal</td>
<td>Pred</td>
<td>Preconditions: Pred meets the Cpp17CopyConstructible requirements. Pred is a binary predicate that takes two arguments of type Key. Pred is an equivalence relation.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::local_iterator</td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as X::iterator's.</td>
<td>A local_iterator object may be used to iterate through a single bucket, but may not be used to iterate across buckets.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::const_local_iterator</td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as X::const_iterator's.</td>
<td>A const_local_iterator object may be used to iterate through a single bucket, but may not be used to iterate across buckets.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::node_type</td>
<td>a specialization of a node-handle class template, such that the public nested types are the same types as the corresponding types in X.</td>
<td>see 22.2.4</td>
<td>compile time</td>
</tr>
<tr>
<td>X(n, hf, eq)</td>
<td>X</td>
<td>Effects: Constructs an empty container with at least n buckets, using hf as the hash function and eq as the key equality predicate.</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>X a(n, hf, eq);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 81: Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X(n, hf)$</td>
<td>$X$</td>
<td><strong>Preconditions:</strong> key_equal meets the Cpp17DefaultConstructible requirements. <strong>Effects:</strong> Constructs an empty container with at least $n$ buckets, using $hf$ as the hash function and key_equal() as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$X(a(n, hf);$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X(n)$</td>
<td>$X$</td>
<td><strong>Preconditions:</strong> hasher and key_equal meet the Cpp17DefaultConstructible requirements. <strong>Effects:</strong> Constructs an empty container with at least $n$ buckets, using hasher() as the hash function and key_equal() as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$X(a(n);$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X()$</td>
<td>$X$</td>
<td><strong>Preconditions:</strong> hasher and key_equal meet the Cpp17DefaultConstructible requirements. <strong>Effects:</strong> Constructs an empty container with an unspecified number of buckets, using hasher() as the hash function and key_equal() as the key equality predicate.</td>
<td>constant</td>
</tr>
<tr>
<td>$X(a);$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X(i, j, n, hf, eq)$</td>
<td>$X$</td>
<td><strong>Preconditions:</strong> value_type is Cpp17EmplaceConstructible into $X$ from *i. <strong>Effects:</strong> Constructs an empty container with at least $n$ buckets, using $hf$ as the hash function and eq as the key equality predicate, and inserts elements from $[i, j)$ into it.</td>
<td>Average case $O(N) \ (N \text{ is distance}(i, j))$, worst case $O(N^2)$</td>
</tr>
<tr>
<td>$X(a(i, j, n, hf, eq);$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X(i, j, n, hf)$</td>
<td>$X$</td>
<td><strong>Preconditions:</strong> key_equal meets the Cpp17DefaultConstructible requirements. value_type is Cpp17EmplaceConstructible into $X$ from *i. <strong>Effects:</strong> Constructs an empty container with at least $n$ buckets, using $hf$ as the hash function and key_equal() as the key equality predicate, and inserts elements from $[i, j)$ into it.</td>
<td>Average case $O(N) \ (N \text{ is distance}(i, j))$, worst case $O(N^2)$</td>
</tr>
<tr>
<td>$X(a(i, j, n, hf);$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Assertion/note</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>X(i, j, n)</td>
<td>X</td>
<td><strong>Preconditions:</strong> hasher and key_equal meet the Cpp17DefaultConstructible requirements. value_type is Cpp17EmplaceConstructible into X from *i. <strong>Effects:</strong> Constructs an empty container with at least n buckets, using hasher() as the hash function and key_equal() as the key equality predicate, and inserts elements from [i, j) into it.</td>
<td></td>
</tr>
<tr>
<td>X a(i, j, n);</td>
<td>X</td>
<td>Average case O(N) (N is distance(i, j)), worst case O(N^2)</td>
<td></td>
</tr>
<tr>
<td>X(i, j)</td>
<td>X</td>
<td><strong>Preconditions:</strong> hasher and key_equal meet the Cpp17DefaultConstructible requirements. value_type is Cpp17EmplaceConstructible into X from *i. <strong>Effects:</strong> Constructs an empty container with an unspecified number of buckets, using hasher() as the hash function and key_equal() as the key equality predicate, and inserts elements from [i, j) into it.</td>
<td></td>
</tr>
<tr>
<td>X a(i, j);</td>
<td>X</td>
<td>Average case O(N) (N is distance(i, j)), worst case O(N^2)</td>
<td></td>
</tr>
<tr>
<td>X(il)</td>
<td>X</td>
<td>Same as X(il.begin(), il.end()).</td>
<td></td>
</tr>
<tr>
<td>X(il, n)</td>
<td>X</td>
<td>Same as X(il.begin(), il.end(), n).</td>
<td></td>
</tr>
<tr>
<td>X(il, n, hf)</td>
<td>X</td>
<td>Same as X(il.begin(), il.end(), n, hf).</td>
<td></td>
</tr>
<tr>
<td>X(il, n, hf, eq)</td>
<td>X</td>
<td>Same as X(il.begin(), il.end(), n, hf, eq).</td>
<td></td>
</tr>
<tr>
<td>X(b)</td>
<td>X</td>
<td><strong>Copy constructor.</strong> In addition to the requirements of Table 73, copies the hash function, predicate, and maximum load factor. <strong>Complexity:</strong> Average case linear in b.size(), worst case quadratic.</td>
<td></td>
</tr>
<tr>
<td>X a(b);</td>
<td>X</td>
<td><strong>Copy assignment operator.</strong> In addition to the requirements of Table 73, copies the hash function, predicate, and maximum load factor. <strong>Complexity:</strong> Average case linear in b.size(), worst case quadratic.</td>
<td></td>
</tr>
</tbody>
</table>
Table 81: Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = il</td>
<td>X&amp;</td>
<td><em>Preconditions</em>: <code>value_type</code> is <code>Cpp17CopyInsertable</code> into <code>X</code> and <code>Cpp17CopyAssignable</code>. Effects: Assigns the range <code>{il.begin(), il.end()}</code> into a. All existing elements of a are either assigned to or destroyed.</td>
<td>Same as <code>a = X(il)</code>.</td>
</tr>
<tr>
<td>b.hash_function()</td>
<td>hasher</td>
<td>Returns: b’s hash function.</td>
<td>constant</td>
</tr>
<tr>
<td>b.key_eq()</td>
<td>key_equal</td>
<td>Returns: b’s key equality predicate.</td>
<td>constant</td>
</tr>
<tr>
<td>a_uniq.emplace(args)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td><em>Preconditions</em>: <code>value_type</code> is <code>Cpp17EmplaceConstructible</code> into <code>X</code> from <code>args</code>. Effects: Inserts a <code>value_type</code> object t constructed with <code>std::forward&lt;Args&gt;(args)</code>... if and only if there is no element in the container with key equivalent to the key of t. The <code>bool</code> component of the returned pair is <code>true</code> if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of t.</td>
<td>Average case ( O(1) ), worst case ( O(a_uniq.size()) ).</td>
</tr>
<tr>
<td>a_eq.emplace(args)</td>
<td>iterator</td>
<td><em>Preconditions</em>: <code>value_type</code> is <code>Cpp17EmplaceConstructible</code> into <code>X</code> from <code>args</code>. Effects: Inserts a <code>value_type</code> object t constructed with <code>std::forward&lt;Args&gt;(args)</code>... and returns the iterator pointing to the newly inserted element.</td>
<td>Average case ( O(1) ), worst case ( O(a_eq.size()) ).</td>
</tr>
<tr>
<td>a.emplace_hint(p, args)</td>
<td>iterator</td>
<td><em>Preconditions</em>: <code>value_type</code> is <code>Cpp17EmplaceConstructible</code> into <code>X</code> from <code>args</code>. Effects: Equivalent to <code>a.emplace(std::forward&lt;Args&gt;(args))...</code>. Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The <code>const_iterator p</code> is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td>Average case ( O(1) ), worst case ( O(a.size()) ).</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Assertion/note</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td><strong>a_uniq.insert(t)</strong></td>
<td><code>pair&lt;iterator, bool&gt;</code></td>
<td>Preconditions: If ( t ) is a non-const rvalue, ( \text{value} ) is <code>Cpp17MoveInsertable</code> into ( X ); otherwise, ( \text{value} ) is <code>Cpp17CopyInsertable</code> into ( X ). Effects: Inserts ( t ) if and only if there is no element in the container with key equivalent to the key of ( t ). The bool component of the returned pair indicates whether the insertion takes place, and the iterator component points to the element with key equivalent to the key of ( t ).</td>
<td></td>
</tr>
<tr>
<td><strong>a_eq.insert(t)</strong></td>
<td><code>iterator</code></td>
<td>Preconditions: If ( t ) is a non-const rvalue, ( \text{value} ) is <code>Cpp17MoveInsertable</code> into ( X ); otherwise, ( \text{value} ) is <code>Cpp17CopyInsertable</code> into ( X ). Effects: Inserts ( t ), and returns an iterator pointing to the newly inserted element.</td>
<td></td>
</tr>
<tr>
<td><strong>a.insert(p, t)</strong></td>
<td><code>iterator</code></td>
<td>Preconditions: If ( t ) is a non-const rvalue, ( \text{value} ) is <code>Cpp17MoveInsertable</code> into ( X ); otherwise, ( \text{value} ) is <code>Cpp17CopyInsertable</code> into ( X ). Effects: Equivalent to a.insert(t). Return value is an iterator pointing to the element with the key equivalent to that of ( t ). The iterator ( p ) is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td></td>
</tr>
<tr>
<td><strong>a.insert(i, j)</strong></td>
<td><code>void</code></td>
<td>Preconditions: ( \text{value} ) is <code>Cpp17EmplaceConstructible</code> into ( X ) from ( \ast i ). Neither ( i ) nor ( j ) are iterators into ( a ). Effects: Equivalent to a.insert(t) for each element in ([i, j)].</td>
<td></td>
</tr>
<tr>
<td><strong>a.insert(il)</strong></td>
<td><code>void</code></td>
<td>Preconditions: ( \text{value} ) is <code>Cpp17EmplaceConstructible</code> into ( X ) from ( \ast i ). Neither ( i ) nor ( j ) are iterators into ( a ). Effects: Equivalent to a.insert(t) for each element in ([i, j)].</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Average case ( \Theta(N) ), where ( N ) is distance(( i ), ( j )), worst case ( \Theta(N(a.size() + 1)) ).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Average case ( \Theta(N) ), where ( N ) is distance(( i ), ( j )), worst case ( \Theta(N(a.size() + 1)) ).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Same as a.insert(il.begin(), il.end()).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Same as a.insert(il.begin(), il.end()).</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Assertion/note pre-/post-condition</td>
<td>Complexity</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td><code>a_uniq. insert(nh)</code></td>
<td><code>insert_return_type</code></td>
<td><strong>Preconditions:</strong> nh is empty or <code>a_uniq.get_allocator() == nh.get_allocator()</code>. <strong>Effects:</strong> If nh is empty, has no effect. Otherwise, inserts the element owned by nh if and only if there is no element in the container with a key equivalent to <code>nh.key()</code>. <strong>Postconditions:</strong> If nh is empty, <code>inserted</code> is <code>false</code>, <code>position</code> is <code>end()</code>, and <code>node</code> is empty. Otherwise if the insertion took place, <code>inserted</code> is <code>true</code>, <code>position</code> points to the inserted element, and <code>node</code> is empty; if the insertion failed, <code>inserted</code> is <code>false</code>, <code>node</code> has the previous value of nh, and <code>position</code> points to an element with a key equivalent to <code>nh.key()</code>.</td>
<td>Average case $O(1)$, worst case $O(a_uniq.size())$.</td>
</tr>
<tr>
<td><code>a_eq. insert(nh)</code></td>
<td><code>iterator</code></td>
<td><strong>Preconditions:</strong> nh is empty or <code>a_eq.get_allocator() == nh.get_allocator()</code>. <strong>Effects:</strong> If nh is empty, has no effect and returns <code>a_eq.end()</code>. Otherwise, inserts the element owned by nh and returns an iterator pointing to the newly inserted element. <strong>Postconditions:</strong> nh is empty.</td>
<td>Average case $O(1)$, worst case $O(a_eq.size())$.</td>
</tr>
<tr>
<td><code>a.insert(q, nh)</code></td>
<td><code>iterator</code></td>
<td><strong>Preconditions:</strong> nh is empty or <code>a.get_allocator() == nh.get_allocator()</code>. <strong>Effects:</strong> If nh is empty, has no effect and returns <code>a.end()</code>. Otherwise, inserts the element owned by nh if and only if there is no element with key equivalent to <code>nh.key()</code> in containers with unique keys; always inserts the element owned by nh in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to <code>nh.key()</code>. The iterator q is a hint pointing to where the search should start. Implementations are permitted to ignore the hint. <strong>Postconditions:</strong> nh is empty if insertion succeeds, unchanged if insertion fails.</td>
<td>Average case $O(1)$, worst case $O(a.size())$.</td>
</tr>
</tbody>
</table>
Table 81: Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.extract(k)</td>
<td>node_type</td>
<td><strong>Effects:</strong> Removes an element in the container with key equivalent to k. <strong>Returns:</strong> A node_type owning the element if found, otherwise an empty node_type.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.extract(q)</td>
<td>node_type</td>
<td><strong>Effects:</strong> Removes the element pointed to by q. <strong>Returns:</strong> A node_type owning that element.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
</tbody>
</table>
| a.merge(a2)   | void        | **Preconditions:**  
a.get_allocator() == a2.get_allocator().  
Attempts to extract each element in a2 and insert it into a using the hash function and key equality predicate of a. In containers with unique keys, if there is an element in a with key equivalent to the key of an element from a2, then that element is not extracted from a2.  
**Postconditions:** Pointers and references to the transferred elements of a2 refer to those same elements but as members of a. Iterators referring to the transferred elements and all iterators referring to a will be invalidated, but iterators to elements remaining in a2 will remain valid. | Average case $\mathcal{O}(N)$, where $N$ is a2.size(), worst case $\mathcal{O}(N*a.size() + N)$. |
| a.erase(k)    | size_type   | **Effects:** Erases all elements with key equivalent to k. **Returns:** The number of elements erased. | Average case $\mathcal{O}(a.count(k))$, worst case $\mathcal{O}(a.size())$. |
| a.erase(q)    | iterator    | **Effects:** Erases the element pointed to by q. **Returns:** The iterator immediately following q prior to the erasure. | Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$. |
| a.erase(r)    | iterator    | **Effects:** Erases the element pointed to by r. **Returns:** The iterator immediately following r prior to the erasure. | Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$. |
| a.erase(q1, q2)| iterator  | **Effects:** Erases all elements in the range [q1, q2).  
**Returns:** The iterator immediately following the erased elements prior to the erasure. | Average case linear in distance(q1, q2), worst case $\mathcal{O}(a.size())$. |
Table 81: Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>Effects: Erases all elements in the container.</td>
<td>Linear in ( a.size() ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Postconditions: ( a.empty() ) is true</td>
<td></td>
</tr>
<tr>
<td>b.find(k)</td>
<td>iterator;</td>
<td>Returns: An iterator pointing to an element with key equivalent to ( k ), or ( b.end() ) if no such element exists.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(b.size()) ).</td>
</tr>
<tr>
<td></td>
<td>const_iterator for const b.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a_tran.find(ke)</td>
<td>iterator;</td>
<td>Returns: An iterator pointing to an element with key equivalent to ( ke ), or ( a_tran.end() ) if no such element exists.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(a_tran.size()) ).</td>
</tr>
<tr>
<td></td>
<td>const_iterator for const a_tran.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b.count(k)</td>
<td>size_type</td>
<td>Returns: The number of elements with key equivalent to ( k ).</td>
<td>Average case ( \Theta(b.count(k)) ), worst case ( \Theta(b.size()) ).</td>
</tr>
<tr>
<td>a_tran.count(ke)</td>
<td>size_type</td>
<td>Returns: The number of elements with key equivalent to ( ke ).</td>
<td>Average case ( \Theta(a_tran.count(ke)) ), worst case ( \Theta(a_tran.size()) ).</td>
</tr>
<tr>
<td>b.contains(k)</td>
<td>bool</td>
<td>Effects: Equivalent to ( b.find(k) != b.end() ).</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(b.size()) ).</td>
</tr>
<tr>
<td>a_tran.contains(ke)</td>
<td>bool</td>
<td>Effects: Equivalent to ( a_tran.find(ke) != a_tran.end() )</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(a_tran.size()) ).</td>
</tr>
<tr>
<td>b.equal_range(k)</td>
<td>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for const b.</td>
<td>Returns: A range containing all elements with keys equivalent to ( k ). Returns make_pair(b.end(), b.end()) if no such elements exist.</td>
<td>Average case ( \Theta(b.count(k)) ), worst case ( \Theta(b.size()) ).</td>
</tr>
<tr>
<td>a_tran.equal_range(ke)</td>
<td>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for const a_tran.</td>
<td>Returns: A range containing all elements with keys equivalent to ( ke ). Returns make_pair(a_tran.end(), a_tran.end()) if no such elements exist.</td>
<td>Average case ( \Theta(a_tran.count(ke)) ), worst case ( \Theta(a_tran.size()) ).</td>
</tr>
<tr>
<td>b.bucket_count()</td>
<td>size_type</td>
<td>Returns: The number of buckets that ( b ) contains.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.max_bucket_count()</td>
<td>size_type</td>
<td>Returns: An upper bound on the number of buckets that ( b ) can ever contain.</td>
<td>Constant</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Assertion/note</td>
<td>Complexity</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------</td>
<td>-----------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>b.bucket(k)</td>
<td>size_type</td>
<td><strong>Preconditions:</strong> b.bucket_count() &gt; 0.</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns:</strong> The index of the bucket in which elements with keys equivalent to k would be found, if any such element existed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Postconditions:</strong> The return value shall be in the range [0, b.bucket_count()).</td>
<td></td>
</tr>
<tr>
<td>b.bucket_size(n)</td>
<td>size_type</td>
<td><strong>Preconditions:</strong> n shall be in the range [0, b.bucket_count()).</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns:</strong> The number of elements in the n^{th} bucket.</td>
<td></td>
</tr>
<tr>
<td>b.begin(n)</td>
<td>local_iterator; const_local_iterator for const b.</td>
<td><strong>Preconditions:</strong> n is in the range [0, b.bucket_count()).</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns:</strong> An iterator referring to the first element in the bucket.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the bucket is empty, then b.begin(n) == b.end(n).</td>
<td></td>
</tr>
<tr>
<td>b.end(n)</td>
<td>local_iterator; const_local_iterator for const b.</td>
<td><strong>Preconditions:</strong> n is in the range [0, b.bucket_count()).</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns:</strong> An iterator which is the past-the-end value for the bucket.</td>
<td></td>
</tr>
<tr>
<td>b.cbegin(n)</td>
<td>const_local_iterator</td>
<td><strong>Preconditions:</strong> n shall be in the range [0, b.bucket_count()).</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns:</strong> An iterator referring to the first element in the bucket.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the bucket is empty, then b.cbegin(n) == b.cend(n).</td>
<td></td>
</tr>
<tr>
<td>b.cend(n)</td>
<td>const_local_iterator</td>
<td><strong>Preconditions:</strong> n is in the range [0, b.bucket_count()).</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns:</strong> An iterator which is the past-the-end value for the bucket.</td>
<td></td>
</tr>
<tr>
<td>b.load_factor()</td>
<td>float</td>
<td><strong>Returns:</strong> The average number of elements per bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.max_load_factor()</td>
<td>float</td>
<td><strong>Returns:</strong> A positive number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number.</td>
<td>Constant</td>
</tr>
<tr>
<td>a.max_load_factor(z)</td>
<td>void</td>
<td><strong>Preconditions:</strong> z is positive. May change the container’s maximum load factor, using z as a hint.</td>
<td>Constant</td>
</tr>
<tr>
<td>a.rehash(n)</td>
<td>void</td>
<td><strong>Postconditions:</strong> a.bucket_count() &gt;= a.size() / a.max_load_factor() and a.bucket_count() &gt;= n.</td>
<td>Average case linear in a.size(), worst case quadratic.</td>
</tr>
</tbody>
</table>
Table 81: Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.reserve(n)</td>
<td>void</td>
<td>Same as a.rehash(ceil(n / a.max_load_factor())).</td>
<td>Average case linear in a.size(), worst case quadratic.</td>
</tr>
</tbody>
</table>

12 Two unordered containers a and b compare equal if a.size() == b.size() and, for every equivalent-key group [Ea1, Ea2) obtained from a.equal_range(Ea1), there exists an equivalent-key group [Eb1, Eb2) obtained from b.equal_range(Ea1), such that is_permutation(Ea1, Ea2, Eb1, Eb2) returns true. For unordered_set and unordered_map, the complexity of operator== (i.e., the number of calls to the == operator of the value_type, to the predicate returned by key_eq(), and to the hasher returned by hash_function()) is proportional to N in the average case and to N^2 in the worst case, where N is a.size(). For unordered_multiset and unordered_multimap, the complexity of operator== is proportional to \( \sum E_i^2 \) in the average case and to \( N^2 \) in the worst case, where \( N \) is a.size(), and \( E_i \) is the size of the \( i \)th equivalent-key group in a. However, if the respective elements of each corresponding pair of equivalent-key groups \( E_a \) and \( E_b \) are arranged in the same order (as is commonly the case, e.g., if a and b are unmodified copies of the same container), then the average-case complexity for unordered_multiset and unordered_multimap becomes proportional to \( N \) (but worst-case complexity remains \( \Theta(N^2) \), e.g., for a pathologically bad hash function). The behavior of a program that uses operator== or operator!= on unordered containers is undefined unless the Pred function object has the same behavior for both containers and the equality comparison function for Key is a refinement of the partition into equivalent-key groups produced by Pred.

13 The iterator types iterator and const_iterator of an unordered associative container are of at least the forward iterator category. For unordered associative containers where the key type and value type are the same, both iterator and const_iterator are constant iterators.

14 The insert and emplace members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The erase members shall invalidate only iterators and references to the erased elements, and preserve the relative order of the elements that are not erased.

15 The insert and emplace members shall not affect the validity of iterators if \( (N+n) \leq z \times B \), where \( N \) is the number of elements in the container prior to the insert operation, \( n \) is the number of elements inserted, \( B \) is the container’s bucket count, and \( z \) is the container’s maximum load factor.

16 The extract members invalidate only iterators to the removed element, and preserve the relative order of the elements that are not erased; pointers and references to the removed element remain valid. However, accessing the element through such pointers and references while the element is owned by a node_type is undefined behavior. References and pointers to an element obtained while it is owned by a node_type are invalidated if the element is successfully inserted.

17 The member function templates find, count, equal_range, and contains shall not participate in overload resolution unless the qualified-ids Pred::is_transparent and Hash::is_transparent are both valid and denote types (13.10.3).

18 A deduction guide for an unordered associative container shall not participate in overload resolution if any of the following are true:
   
   (18.1) — It has an InputIterator template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
   
   (18.2) — It has an Allocator template parameter and a type that does not qualify as an allocator is deduced for that parameter.
   
   (18.3) — It has a Hash template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
   
   (18.4) — It has a Pred template parameter and a type that qualifies as an allocator is deduced for that parameter.

230) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.
22.2.7.2 Exception safety guarantees

For unordered associative containers, no `clear()` function throws an exception. The `erase(k)` function does not throw an exception unless that exception is thrown by the container’s `Hash` or `Pred` object (if any).

For unordered associative containers, if an exception is thrown by any operation other than the container's hash function from within a `insert` or `emplace` function inserting a single element, the insertion has no effect.

For unordered associative containers, no `swap` function throws an exception unless that exception is thrown by the swap of the container’s `Hash` or `Pred` object (if any).

For unordered associative containers, if an exception is thrown from within a `rehash()` function other than by the container’s hash function or comparison function, the `rehash()` function has no effect.

22.3 Sequence containers

22.3.1 In general

The headers `<array>` (22.3.2), `<deque>` (22.3.3), `<forward_list>` (22.3.4), `<list>` (22.3.5), and `<vector>` (22.3.6) define class templates that meet the requirements for sequence containers.

The following exposition-only alias template may appear in deduction guides for sequence containers:

```cpp
template<class InputIterator>
using iter_value_type = typename iterator_traits<InputIterator>::value_type; // exposition only
```

22.3.2 Header `<array>` synopsis

```cpp
#include <array>
#include <initializer_list>

namespace std {
    class array;

    template<class T, size_t N> struct array;
    template<class T, size_t N> constexpr bool operator==(const array<T, N>& x, const array<T, N>& y);
    template<class T, size_t N> constexpr synth-three-way-result<T> operator<=>(const array<T, N>& x, const array<T, N>& y);
    template<class T, size_t N> constexpr void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));

    template<class T, size_t N> constexpr array<remove_cv_t<T>, N> to_array(T (&a)[N]);
    template<class T, size_t N> constexpr array<remove_cv_t<T>, N> to_array(T (&&a)[N]);

    template<size_t I, class T, size_t N>
        constexpr T& get(array<T, N>&) noexcept;
    template<size_t I, class T, size_t N>
        constexpr T&& get(array<T, N>&&) noexcept;
    template<size_t I, class T, size_t N>
        constexpr const T& get(const array<T, N>&) noexcept;
    template<size_t I, class T, size_t N>
        constexpr const T&& get(const array<T, N>&&) noexcept;
}
```

§ 22.3.2 840
22.3.3 Header <deque> synopsis

```cpp
#include <compare>  // see 17.11.1
#include <initializer_list>  // see 17.10.2

namespace std {
    // 22.3.8, class template deque
    template<class T, class Allocator = allocator<T>> class deque;

    template<class T, class Allocator>
    bool operator==(const deque<T, Allocator>& x, const deque<T, Allocator>& y);

    template<class T, class Allocator>
    synth-three-way-result<T> operator<=>(const deque<T, Allocator>& x,
                                          const deque<T, Allocator>& y);

    template<class T, class Allocator>
    void swap(deque<T, Allocator>& x, deque<T, Allocator>& y)
        noexcept(noexcept(x.swap(y)));

    template<class T, class Allocator, class U>
    typename deque<T, Allocator>::size_type
        erase(deque<T, Allocator>& c, const U& value);

    template<class T, class Allocator, class Predicate>
    typename deque<T, Allocator>::size_type
        erase_if(deque<T, Allocator>& c, Predicate pred);

    namespace pmr {
        template<class T>
        using deque = std::deque<T, polymorphic_allocator<T>>;
    }
}
```

22.3.4 Header <forward_list> synopsis

```cpp
#include <compare>  // see 17.11.1
#include <initializer_list>  // see 17.10.2

namespace std {
    // 22.3.9, class template forward_list
    template<class T, class Allocator = allocator<T>> class forward_list;

    template<class T, class Allocator>
    bool operator==(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);

    template<class T, class Allocator>
    synth-three-way-result<T> operator<=>(const forward_list<T, Allocator>& x,
                                           const forward_list<T, Allocator>& y);

    template<class T, class Allocator>
    void swap(forward_list<T, Allocator>& x, forward_list<T, Allocator>& y)
        noexcept(noexcept(x.swap(y)));

    template<class T, class Allocator, class U>
    typename forward_list<T, Allocator>::size_type
        erase(forward_list<T, Allocator>& c, const U& value);

    template<class T, class Allocator, class Predicate>
    typename forward_list<T, Allocator>::size_type
        erase_if(forward_list<T, Allocator>& c, Predicate pred);

    namespace pmr {
        template<class T>
        using forward_list = std::forward_list<T, polymorphic_allocator<T>>;
    }
}
```
22.3.5 Header <list> synopsis

```cpp
#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.2

namespace std {
    // 22.3.10, class template list
template<class T, class Allocator = allocator<T>> class list;

template<class T, class Allocator>
bool operator==(const list<T, Allocator>& x, const list<T, Allocator>& y);

template<class T, class Allocator>
synth-three-way-result<
    T> operator<=>(const list<T, Allocator>& x,
    const list<T, Allocator>& y);

template<class T, class Allocator>
void swap(list<T, Allocator>& x, list<T, Allocator>& y)
noexcept(noexcept(x.swap(y))));

namespace pmr {
    template<class T>
        using list = std::list<T, polymorphic_allocator<T>>;
}
}
```

22.3.6 Header <vector> synopsis

```cpp
#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.2

namespace std {
    // 22.3.11, class template vector
template<class T, class Allocator = allocator<T>> class vector;

template<class T, class Allocator>
constexpr bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);

template<class T, class Allocator>
constexpr synth-three-way-result<
    T> operator<=>(const vector<T, Allocator>& x,
    const vector<T, Allocator>& y);

template<class T, class Allocator>
constexpr void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
noexcept(noexcept(x.swap(y))));

namespace pmr {
    template<class T>
        using vector = std::vector<T, polymorphic_allocator<T>>;
}
```
namespace pmr {
    template<class T>
    using vector = std::vector<T, polymorphic_allocator<T>>;
}

22.3.7 Class template array

22.3.7.1 Overview

The header `<array>` defines a class template for storing fixed-size sequences of objects. An array is a contiguous container (22.2.1). An instance of `array<T, N>` stores N elements of type T, so that `size() == N` is an invariant.

An array is an aggregate (9.4.2) that can be list-initialized with up to N elements whose types are convertible to T.

An array meets all of the requirements of a container and of a reversible container (22.2), except that a default constructed array object is not empty and that swap does not have constant complexity. An array meets some of the requirements of a sequence container (22.2.3). Descriptions are provided here only for operations on array that are not described in one of these tables and for operations where there is additional semantic information.

array<T, N> is a structural type (13.2) if T is a structural type. Two values a1 and a2 of type array<T, N> are template-argument-equivalent (13.6) if and only if each pair of corresponding elements in a1 and a2 are template-argument-equivalent.

The types iterator and const_iterator meet the constexpr iterator requirements (23.3.1).

namespace std {
    template<class T, size_t N>
    struct array {
        // types
        using value_type = T;
        using pointer = T*;
        using const_pointer = const T*;
        using reference = T&;
        using const_reference = const T&;
        using size_type = size_t;
        using difference_type = ptrdiff_t;
        using iterator = implementation-defined; // see 22.2
        using const_iterator = implementation-defined; // see 22.2
        using reverse_iterator = std::reverse_iterator<iterator>;
        using const_reverse_iterator = std::reverse_iterator<const_iterator>;

        // no explicit construct/copy/destroy for aggregate type
        constexpr void fill(const T& u);
        constexpr void swap(array&) noexcept(is_nothrow_swappable_v<T>);

        // iterators
        constexpr iterator begin() noexcept;
        constexpr const_iterator begin() const noexcept;
        constexpr iterator end() noexcept;
        constexpr const_iterator end() const noexcept;
        constexpr reverse_iterator rbegin() noexcept;
        constexpr const_reverse_iterator rbegin() const noexcept;
        constexpr reverse_iterator rend() noexcept;
        constexpr const_reverse_iterator rend() const noexcept;
        constexpr const_iterator cbegin() const noexcept;
        constexpr const_iterator cend() const noexcept;
        constexpr const_reverse_iterator crbegin() const noexcept;
        constexpr const_reverse_iterator crend() const noexcept;
    }
};

§ 22.3.7.1
// capacity
[[nodiscard]] constexpr bool empty() const noexcept;
constexpr size_type size() const noexcept;
constexpr size_type max_size() const noexcept;

// element access
constexpr reference operator[](size_type n);
constexpr const_reference operator[](size_type n) const;
constexpr reference at(size_type n);
constexpr const_reference at(size_type n) const;
constexpr reference front();
constexpr const_reference front() const;
constexpr reference back();
constexpr const_reference back() const;

constexpr T * data() noexcept;
constexpr const T * data() const noexcept;

};

// template

template<class T, class... U>
array(T, U ...) -> array<T, 1 + sizeof...(U)>;

22.3.7.2 Constructors, copy, and assignment [array.cons]

1 The conditions for an aggregate (9.4.2) shall be met. Class array relies on the implicitly-declared special
member functions (11.4.5.2, 11.4.7, and 11.4.5.3) to conform to the container requirements table in 22.2. In
addition to the requirements specified in the container requirements table, the implicit move constructor and
move assignment operator for array require that T be Cpp17MoveConstructible or Cpp17MoveAssignable,
respectively.

template<class T, class... U>
array(T, U ...) -> array<T, 1 + sizeof...(U)>;

2 Mandates: (is_same_v<T, U> && ...) is true.

22.3.7.3 Member functions [array.members]

constexpr size_type size() const noexcept;

1 Returns: N.

constexpr T* data() noexcept;
constexpr const T* data() const noexcept;

2 Returns: A pointer such that [data(), data() + size()) is a valid range. For a non-empty array,
data() == addressof(front()).

constexpr void fill(const T& u);

3 Effects: As if by fill_n(begin(), N, u).

constexpr void swap(array& y) noexcept(is_nothrow_swappable_v<T>);

4 Effects: Equivalent to swap_ranges(begin(), end(), y.begin()).

5 [Note 1: Unlike the swap function for other containers, array::swap takes linear time, can exit via an exception,
and does not cause iterators to become associated with the other container. —end note]

22.3.7.4 Specialized algorithms [array.special]

template<class T, size_t N>
constexpr void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));

1 Constraints: N == 0 or is_swappable_v<T> is true.

2 Effects: As if by x.swap(y).

3 Complexity: Linear in N.
22.3.7.5 Zero-sized arrays 

array shall provide support for the special case \( N == 0 \).

In the case that \( N == 0 \), \begin{array} \text{begin}() == \text{end}() \end{array} \text{ is unique value. The return value of data()} \text{ is unspecified.}

The effect of calling \begin{array} \text{front}() \text{ or back}() \end{array} \text{ for a zero-sized array is undefined.}

Member function \text{swap()} \text{ shall have a non-throwing exception specification.}

22.3.7.6 Array creation functions

```cpp
template<class T, size_t N>
constexpr array<remove_cv_t<T>, N> to_array(T (&a)[N]);
```

Mandates: is_array_v<T> is false and is_constructible_v<T, T&> is true.
Preconditions: T meets the Cpp17CopyConstructible requirements.
Returns: {{ a[0], ..., a[N - 1] }}.

```cpp
template<class T, size_t N>
constexpr array<remove_cv_t<T>, N> to_array(T (&&a)[N]);
```

Mandates: is_array_v<T> is false and is_move_constructible_v<T> is true.
Preconditions: T meets the Cpp17MoveConstructible requirements.
Returns: {{ std::move(a[0]), ..., std::move(a[N - 1]) }}.

22.3.7.7 Tuple interface

```cpp
template<class T, size_t N>
struct tuple_size<array<T, N>> : integral_constant<size_t, N> { };
```

```cpp
template<size_t I, class T, size_t N>
struct tuple_element<I, array<T, N>> {
    using type = T;
};
```

Mandates: \( I < N \) is true.

```cpp
template<size_t I, class T, size_t N>
constexpr T& get(array<T, N>& a) noexcept;
template<size_t I, class T, size_t N>
constexpr T&& get(array<T, N>&& a) noexcept;
```

Mandates: \( I < N \) is true.
Returns: A reference to the \( I \)th element of a, where indexing is zero-based.

22.3.8 Class template deque

22.3.8.1 Overview

A deque is a sequence container that supports random access iterators (23.3.5.7). In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. That is, a deque is especially optimized for pushing and popping elements at the beginning and end. Storage management is handled automatically.

A deque meets all of the requirements of a container, of a reversible container (given in tables in 22.2), of a sequence container, including the optional sequence container requirements (22.2.3), and of an allocator-aware container (Table 76). Descriptions are provided here only for operations on deque that are not described in one of these tables or for operations where there is additional semantic information.

```cpp
namespace std {
    template<class T, class Allocator = allocator<T>>
    class deque {
        public:
            // types
```
using value_type = T;
using pointer = typename allocator_traits<Allocator>::pointer;
using const_pointer = typename allocator_traits<Allocator>::const_pointer;
using reference = value_type&;
using const_reference = const value_type&;
using size_type = implementation-defined; // see 22.2
using difference_type = implementation-defined; // see 22.2
using iterator = implementation-defined; // see 22.2
using const_iterator = implementation-defined; // see 22.2
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;

// 22.3.8.2, construct/copy/destroy
deqe() : deque(Allocator()) {} explicit deque(const Allocator&);
deqe(size_type n, const Allocator& = Allocator());
template<class InputIterator>
deqe(InputIterator first, InputIterator last, const Allocator& = Allocator());
deqe(const deque& x);
deqe(deque&);
deqe(const deque&, const Allocator&);
deqe(deque&, const Allocator&);
deqe(initializer_list<T>, const Allocator& = Allocator());
~deque();
deque& operator=(const deque& x);
deque& operator=(deque& x);

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;

// element access
reference operator[](size_type n);
const_reference operator[](size_type n) const;
reference at(size_type n);
const_reference at(size_type n) const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// 22.3.8.4, modifiers
template<class... Args> reference emplace_front(Args&&... args);
template<class... Args> reference emplace_back(Args&&... args);
template<class... Args> iterator emplace(const_iterator position, Args&&... args);

void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);

iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template<class InputIterator>
iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);

void pop_front();
void pop_back();

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);

void swap(deque&) noexcept(allocator_traits<Allocator>::is_always_equal::value);
void clear() noexcept;

// swap
template<class T, class Allocator>
void swap(deque<T, Allocator>& x, deque<T, Allocator>& y) noexcept(x.swap(y));

22.3.8.2 Constructors, copy, and assignment [deque.cons]

explicit deque(const Allocator&);

Effects: Constructs an empty deque, using the specified allocator.
Complexity: Constant.

explicit deque(size_type n, const Allocator& = Allocator());

Preconditions: T is Cpp17DefaultInsertable into *this.
Effects: Constructs a deque with n default-inserted elements using the specified allocator.
Complexity: Linear in n.

deque(size_type n, const T& value, const Allocator& = Allocator());

Preconditions: T is Cpp17CopyInsertable into *this.
Effects: Constructs a deque with n copies of value, using the specified allocator.
Complexity: Linear in n.
template<class InputIterator>
deque(InputIterator first, InputIterator last, const Allocator& = Allocator());

Effects: Constructs a deque equal to the range [first, last), using the specified allocator.

Complexity: Linear in distance(first, last).

22.3.8.3 Capacity

void resize(size_type sz);

Preconditions: T is Cpp17MoveInsertable and Cpp17DefaultInsertable into *this.

Effects: If sz < size(), erases the last size() - sz elements from the sequence. Otherwise, appends sz - size() default-inserted elements to the sequence.

void resize(size_type sz, const T& c);

Preconditions: T is Cpp17CopyInsertable into *this.

Effects: If sz < size(), erases the last size() - sz elements from the sequence. Otherwise, appends sz - size() copies of c to the sequence.

void shrink_to_fit();

Preconditions: T is Cpp17MoveInsertable into *this.

Effects: shrink_to_fit is a non-binding request to reduce memory use but does not change the size of the sequence.

[Note 1: The request is non-binding to allow latitude for implementation-specific optimizations. — end note]

If the size is equal to the old capacity, or if an exception is thrown other than by the move constructor of a non-Cpp17CopyInsertable T, then there are no effects.

Complexity: If the size is not equal to the old capacity, linear in the size of the sequence; otherwise constant.

Remarks: If the size is not equal to the old capacity, then invalidates all the references, pointers, and iterators referring to the elements in the sequence, as well as the past-the-end iterator.

22.3.8.4 Modifiers

iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);

template<class InputIterator>
iterator insert(const_iterator position,
               InputIterator first, InputIterator last);

iterator insert(const_iterator position, initializer_list<T>);

template<class... Args> reference emplace_front(Args&&... args);

template<class... Args> reference emplace_back(Args&&... args);

template<class... Args> iterator emplace(const_iterator position, Args&&... args);

void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);

Effects: An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque invalidates all the iterators to the deque, but has no effect on the validity of references to elements of the deque.

Complexity: The complexity is linear in the number of elements inserted plus the lesser of the distances to the beginning and end of the deque. Inserting a single element at either the beginning or end of a deque always takes constant time and causes a single call to a constructor of T.

Remarks: If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of T there are no effects. If an exception is thrown while inserting a single element at either end, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-Cpp17CopyInsertable T, the effects are unspecified.
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_front();
void pop_back();

Effects: An erase operation that erases the last element of a deque invalidates only the past-the-end iterator and all iterators and references to the erased elements. An erase operation that erases the first element of a deque but not the last element invalidates only iterators and references to the erased elements. An erase operation that erases neither the first element nor the last element of a deque invalidates the past-the-end iterator and all iterators and references to all the elements of the deque.

[Note 1: pop_front and pop_back are erase operations. — end note]

Throws: Nothing unless an exception is thrown by the assignment operator of T.

Complexity: The number of calls to the destructor of T is the same as the number of elements erased, but the number of calls to the assignment operator of T is no more than the lesser of the number of elements before the erased elements and the number of elements after the erased elements.

22.3.8.5 Erasure

template<class T, class Allocator, class U>
  typename deque<T, Allocator>::size_type
  erase(deque<T, Allocator>& c, const U& value);

Effects: Equivalent to:
  auto it = remove(c.begin(), c.end(), value);
  auto r = distance(it, c.end());
  c.erase(it, c.end());
  return r;

template<class T, class Allocator, class Predicate>
  typename deque<T, Allocator>::size_type
  erase_if(deque<T, Allocator>& c, Predicate pred);

Effects: Equivalent to:
  auto it = remove_if(c.begin(), c.end(), pred);
  auto r = distance(it, c.end());
  c.erase(it, c.end());
  return r;

22.3.9 Class template forward_list

22.3.9.1 Overview

A forward_list is a container that supports forward iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Fast random access to list elements is not supported.

[Note 1: It is intended that forward_list have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted. — end note]

A forward_list meets all of the requirements of a container (Table 73), except that the size() member function is not provided and operator== has linear complexity. A forward_list also meets all of the requirements for an allocator-aware container (Table 76). In addition, a forward_list provides the assign member functions (Table 77) and several of the optional container requirements (Table 78). Descriptions are provided here only for operations on forward_list that are not described in that table or for operations where there is additional semantic information.

[Note 2: Modifying any list requires access to the element preceding the first element of interest, but in a forward_list there is no constant-time way to access a preceding element. For this reason, erase_after and splice_after take fully-open ranges, not semi-open ranges. — end note]
using allocator_type = Allocator;
using pointer = typename allocator_traits<Allocator>::pointer;
using const_pointer = typename allocator_traits<Allocator>::const_pointer;
using reference = value_type&;
using const_reference = const value_type&;
using size_type = implementation-defined;  // see 22.2
using difference_type = implementation-defined;  // see 22.2
using iterator = implementation-defined;  // see 22.2
using const_iterator = implementation-defined;  // see 22.2

// 22.3.9.2, construct/copy/destroy
forward_list() : forward_list(Allocator()) { }
explicit forward_list(const Allocator&);
explicit forward_list(size_type n, const Allocator& = Allocator());
forward_list(size_type n, const T& value, const Allocator& = Allocator());
template<class InputIterator>
forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
forward_list(forward_list&& x);
forward_list(const forward_list& x);
forward_list(forward_list&& x);
forward_list(initializer_list<T>, const Allocator& = Allocator());
~forward_list();
forward_list& operator=(const forward_list& x);
forward_list& operator=(forward_list&& x) noexcept(allocator_traits<Allocator>::is_always_equal::value);
forward_list& operator=(initializer_list<T>);
template<class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;

// 22.3.9.3, iterators
iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cbefore_begin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type max_size() const noexcept;

// 22.3.9.4, element access
reference front();
const_reference front() const;

// 22.3.9.5, modifiers
template<class... Args> reference emplace_front(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void pop_front();
template<class... Args> iterator emplace_after(const_iterator position, Args&&... args);
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);
iterator insert_after(const_iterator position, size_type n, const T& x);
    template<class InputIterator>
    iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
iterator insert_after(const_iterator position, initializer_list<T> il);
iterator erase_after(const_iterator position);
iterator erase_after(const_iterator position, const_iterator last);
void swap(forward_list&) noexcept(allocator_traits<Allocator>::is_always_equal::value);
void resize(size_type sz);
void resize(size_type sz, const value_type& c);
void clear() noexcept;

// 22.3.9.6, forward_list operations
void splice_after(const_iterator position, forward_list& x);
void splice_after(const_iterator position, forward_list&& x);
void splice_after(const_iterator position, forward_list& x, const_iterator i);
void splice_after(const_iterator position, forward_list&& x, const_iterator i);
void splice_after(const_iterator position, forward_list& x, const_iterator first, const_iterator last);
void splice_after(const_iterator position, forward_list&& x, const_iterator first, const_iterator last);

size_type remove(const T& value);
template<class Predicate> size_type remove_if(Predicate pred);

size_type unique();
template<class BinaryPredicate> size_type unique(BinaryPredicate binary_pred);

void merge(forward_list& x);
void merge(forward_list&& x);
template<class Compare> void merge(forward_list& x, Compare comp);
template<class Compare> void merge(forward_list&& x, Compare comp);

void sort();
template<class Compare> void sort(Compare comp);

void reverse() noexcept;
};

template<class InputIterator, class Allocator = allocator<
    iter-value-type<InputIterator>>>
forward_list<InputIterator, InputIterator, Allocator = Allocator>()
    -> forward_list<iter-value-type<InputIterator>, Allocator>;

// swap
template<class T, class Allocator>
void swap(forward_list<T, Allocator>& x, forward_list<T, Allocator>& y)
    noexcept(noexcept(x.swap(y)));

An incomplete type T may be used when instantiating forward_list if the allocator meets the allocator completeness requirements (16.4.4.6.2). T shall be complete before any member of the resulting specialization of forward_list is referenced.

22.3.9.2 Constructors, copy, and assignment [forwardlist.cons]

explicit forward_list(const Allocator&);

Effects: Constructs an empty forward_list object using the specified allocator.

Complexity: Constant.

explicit forward_list(size_type n, const Allocator& = Allocator());

Preconditions: T is Cpp17DefaultInsertable into *this.

§ 22.3.9.2
Effects: Constructs a `forward_list` object with \( n \) default-inserted elements using the specified allocator.

Complexity: Linear in \( n \).

```cpp
forward_list(size_type n, const T& value, const Allocator& = Allocator());
```

Preconditions: `T` is `Cpp17CopyInsertable` into `*this`.

Effects: Constructs a `forward_list` object with \( n \) copies of `value` using the specified allocator.

Complexity: Linear in \( n \).

```cpp
template<class InputIterator>
forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
```

Effects: Constructs a `forward_list` object equal to the range `[first, last)`.

Complexity: Linear in `distance(first, last)`.

### 22.3.9.3 Iterators

- `iterator before_begin()` noexcept;
- `const_iterator before_begin()` const noexcept;
- `const_iterator cbefore_begin()` const noexcept;

**Effects:** `cbefore_begin()` is equivalent to `const_cast<forward_list const&>(*this).before_begin()`.

**Returns:** A non-dereferenceable iterator that, when incremented, is equal to the iterator returned by `begin()`.

**Remarks:** `before_begin() == end()` shall equal false.

### 22.3.9.4 Element access

- `reference front();`
- `const_reference front() const;`

**Returns:** `*begin()`.

### 22.3.9.5 Modifiers

None of the overloads of `insert_after` shall affect the validity of iterators and references, and `erase_after` shall invalidate only iterators and references to the erased elements. If an exception is thrown during `insert_after` there shall be no effect. Inserting \( n \) elements into a `forward_list` is linear in \( n \), and the number of calls to the copy or move constructor of \( T \) is exactly equal to \( n \). Erasing \( n \) elements from a `forward_list` is linear in \( n \) and the number of calls to the destructor of type \( T \) is exactly equal to \( n \).

```cpp
template<class... Args> reference emplace_front(Args&&... args);
```

**Effects:** Inserts an object of type `value_type` constructed with `value_type(std::forward<Args>(args)...)` at the beginning of the list.

```cpp
void push_front(const T& x);
void push_front(T&& x);
```

**Effects:** Inserts a copy of `x` at the beginning of the list.

```cpp
void pop_front();
```

**Effects:** As if by `erase_after(before_begin())`.

```cpp
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);
```

**Preconditions:** `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`.

**Effects:** Inserts a copy of `x` after `position`.

**Returns:** An iterator pointing to the copy of `x`. 

§ 22.3.9.5 852
iterator insert_after(const_iterator position, size_type n, const T& x);

Preconditions: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).

Effects: Inserts n copies of x after position.

Returns: An iterator pointing to the last inserted copy of x or position if n == 0.

```cpp
template<class InputIterator>
iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
```

Preconditions: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()). Neither first nor last are iterators in *this.

Effects: Inserts copies of elements in [first, last) after position.

Returns: An iterator pointing to the last inserted element or position if first == last.

```cpp
iterator insert_after(const_iterator position, initializer_list<T> il);
```

Effects: insert_after(p, il.begin(), il.end()).

 Returns: An iterator pointing to the last inserted element or position if il is empty.

```cpp
template<class... Args>
iterator emplace_after(const_iterator position, Args&&... args);
```

Preconditions: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).

Effects: Inserts an object of type value_type constructed with value_type(std::forward<Args>(args))... after position.

Returns: An iterator pointing to the new object.

```cpp
iterator erase_after(const_iterator position);
```

Preconditions: The iterator following position is dereferenceable.

Effects: Erases the element pointed to by the iterator following position.

Returns: An iterator pointing to the element following the one that was erased, or end() if no such element exists.

Throws: Nothing.

```cpp
iterator erase_after(const_iterator position, const_iterator last);
```

Preconditions: All iterators in the range (position, last) are dereferenceable.

Effects: Erases the elements in the range (position, last).

Returns: last.

Throws: Nothing.

```cpp
void resize(size_type sz);
```

Preconditions: T is Cpp17DefaultInsertable into *this.

Effects: If sz < distance(begin(), end()), erases the last distance(begin(), end()) - sz elements from the list. Otherwise, inserts sz - distance(begin(), end()) default-inserted elements at the end of the list.

```cpp
void resize(size_type sz, const value_type& c);
```

Preconditions: T is Cpp17CopyInsertable into *this.

Effects: If sz < distance(begin(), end()), erases the last distance(begin(), end()) - sz elements from the list. Otherwise, inserts sz - distance(begin(), end()) copies of c at the end of the list.

```cpp
void clear() noexcept;
```

Effects: Erases all elements in the range [begin(), end()).

Remarks: Does not invalidate past-the-end iterators.
22.3.9.6 Operations

In this subclause, arguments for a template parameter named `Predicate` or `BinaryPredicate` shall meet the corresponding requirements in 25.2. For `merge` and `sort`, the definitions and requirements in 25.8 apply.

```cpp
void splice_after(const_iterator position, forward_list& x);
void splice_after(const_iterator position, forward_list&& x);
```

Preconditions: `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`. `get_allocator()` == `x.get_allocator()` is true. `addressof(x) != this` is true.

Effects: Inserts the contents of `x` after `position`, and `x` becomes empty. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

Throws: Nothing.

Complexity: $\Theta(1)$

```cpp
void splice_after(const_iterator position, forward_list& x, const_iterator i);
void splice_after(const_iterator position, forward_list&& x, const_iterator i);
```

Preconditions: `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`. The iterator following `i` is a dereferenceable iterator in `x`. `get_allocator()` == `x.get_allocator()` is true.

Effects: Inserts the element following `i` into `*this`, following `position`, and removes it from `x`. The result is unchanged if `position == i` or `position == ++i`. Pointers and references to `++i` continue to refer to the same element but as a member of `*this`. Iterators to `++i` continue to refer to the same element, but now behave as iterators into `*this`, not into `x`.

Throws: Nothing.

Complexity: $\Theta(1)$

```cpp
void splice_after(const_iterator position, forward_list& x, const_iterator first, const_iterator last);
void splice_after(const_iterator position, forward_list&& x, const_iterator first, const_iterator last);
```

Preconditions: `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`. `(first, last)` is a valid range in `x`, and all iterators in the range `(first, last)` are dereferenceable. `position` is not an iterator in the range `(first, last)`. `get_allocator()` == `x.get_allocator()` is true.

Effects: Inserts elements in the range `(first, last)` after `position` and removes the elements from `x`. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

Complexity: $\Theta(distance(first, last))$

```cpp
size_type remove(const T& value);
template<class Predicate> size_type remove_if(Predicate pred);
```

Effects: Erases all the elements in the list referred to by a list iterator `i` for which the following conditions hold: `*i == value` (for `remove()`), `pred(*i)` is `true` (for `remove_if()`). Invalidates only the iterators and references to the erased elements.

Returns: The number of elements erased.

Throws: Nothing unless an exception is thrown by the equality comparison or the predicate.

Complexity: Exactly `distance(begin(), end())` applications of the corresponding predicate.

Remarks: Stable (16.4.6.8).

```cpp
size_type unique();
template<class BinaryPredicate> size_type unique(BinaryPredicate pred);
```

Effects: Erases all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first + 1, last)` for which `*i == *(i-1)` (for the version with no
arguments) or pred(*i, *(i - 1)) (for the version with a predicate argument) holds. Invalidates only the iterators and references to the erased elements.

Returns: The number of elements erased.

Throws: Nothing unless an exception is thrown by the equality comparison or the predicate.

Complexity: If the range [first, last) is not empty, exactly (last - first) - 1 applications of the corresponding predicate, otherwise no applications of the predicate.

```cpp
void merge(forward_list& x);
void merge(forward_list&& x);
template<class Compare> void merge(forward_list& x, Compare comp);
template<class Compare> void merge(forward_list&& x, Compare comp);
```

Preconditions: *this and x are both sorted with respect to the comparator operator< (for the first two overloads) or comp (for the last two overloads), and get_allocator() == x.get_allocator() is true.

Effects: Merges the two sorted ranges [begin(), end()) and [x.begin(), x.end()). x is empty after the merge. If an exception is thrown other than by a comparison there are no effects. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

Complexity: At most distance(begin(), end()) + distance(x.begin(), x.end()) - 1 comparisons.

Remarks: Stable (16.4.6.8).

```cpp
void sort();
template<class Compare> void sort(Compare comp);
```

Effects: Sorts the list according to the operator< or the comp function object. If an exception is thrown, the order of the elements in *this is unspecified. Does not affect the validity of iterators and references.

Complexity: Approximately \( N \log N \) comparisons, where \( N \) is distance(begin(), end()).

Remarks: Stable (16.4.6.8).

```cpp
void reverse() noexcept;
```

Effects: Reverses the order of the elements in the list. Does not affect the validity of iterators and references.

Complexity: Linear time.

### 22.3.9.7 Erasure

```cpp
template<class T, class Allocator, class U>
typename forward_list<T, Allocator>::size_type
erase(forward_list<T, Allocator>& c, const U& value);
```

Effects: Equivalent to: `return erase_if(c, [&] (auto& elem) { return elem == value; });`

```cpp
template<class T, class Allocator, class Predicate>
typename forward_list<T, Allocator>::size_type
erase_if(forward_list<T, Allocator>& c, Predicate pred);
```

Effects: Equivalent to: `return c.remove_if(pred);`

### 22.3.10 Class template list

#### 22.3.10.1 Overview

A list is a sequence container that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors (22.3.11) and deques (22.3.8), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

A list meets all of the requirements of a container, of a reversible container (given in two tables in 22.2), of a sequence container, including most of the optional sequence container requirements (22.2.3), and of an
allocator-aware container (Table 76). The exceptions are the operator[] and at member functions, which are not provided.\textsuperscript{231} Descriptions are provided here only for operations on list that are not described in one of these tables or for operations where there is additional semantic information.

namespace std {

    template<class T, class Allocator = allocator<T>>
    class list {
        public:
            // types
            using value_type = T;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation-defined; // see 22.2
            using difference_type = implementation-defined; // see 22.2
            using iterator = implementation-defined; // see 22.2
            using const_iterator = implementation-defined; // see 22.2
            using reverse_iterator = std::reverse_iterator<iterator>;
            using const_reverse_iterator = std::reverse_iterator<const_iterator>;
            // 22.3.10.2, construct/copy/destroy
            list() : list(Allocator()) { } explicit list(const Allocator&);
            explicit list(size_type n, const Allocator& = Allocator());
            list(size_type n, const T& value, const Allocator& = Allocator());
            template<class InputIterator>
            list(InputIterator first, InputIterator last, const Allocator& = Allocator());
            list(const list& x);
            list(list&& x);
            list(const list&, const Allocator&);
            list(list&&, const Allocator&);
            list(initializer_list<T>, const Allocator& = Allocator());
            ~list();
            list& operator=(const list& x);
            list& operator=(list&& x) noexcept(allocator_traits<Allocator>::is_always_equal::value);
            list& operator=(initializer_list<T>);
            template<class InputIterator>
            void assign(InputIterator first, InputIterator last);
            void assign(size_type n, const T& t);
            void assign(initializer_list<T>);
            allocator_type get_allocator() const noexcept;
            // iterators
            iterator begin() noexcept;
            const_iterator begin() const noexcept;
            iterator end() noexcept;
            const_iterator end() const noexcept;
            reverse_iterator rbegin() noexcept;
            const_reverse_iterator rbegin() const noexcept;
            reverse_iterator rend() noexcept;
            const_reverse_iterator rend() const noexcept;
            const_iterator cbegin() const noexcept;
            const_iterator cend() const noexcept;
            const_reverse_iterator crbegin() const noexcept;
            const_reverse_iterator crend() const noexcept;
            // 22.3.10.3, capacity
            [[nodiscard]] bool empty() const noexcept;
    }

\textsuperscript{231} These member functions are only provided by containers whose iterators are random access iterators.
size_type size() const noexcept;
size_type max_size() const noexcept;
void resize(size_type sz);
void resize(size_type sz, const T& c);

// element access
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// 22.3.10.4, modifiers
template<class... Args> reference emplace_front(Args&&... args);
template<class... Args> reference emplace_back(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void pop_front();
void push_back(const T& x);
void push_back(T&& x);
void pop_back();

template<class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template<class InputIterator>
iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, const_iterator position, initializer_list<T> il);

iterator erase(const_iterator position);
iterator erase(const_iterator position, const_iterator last);
void swap(list&) noexcept(allocator_traits<Allocator>::is_always_equal::value);
void clear() noexcept;

// 22.3.10.5, list operations
void splice(const_iterator position, list& x);
void splice(const_iterator position, list&& x);
void splice(const_iterator position, list& x, const_iterator i);
void splice(const_iterator position, list&& x, const_iterator i);
void splice(const_iterator position, list& x, const_iterator first, const_iterator last);
void splice(const_iterator position, list&& x, const_iterator first, const_iterator last);

size_type remove(const T& value);
template<class Predicate> size_type remove_if(Predicate pred);

size_type unique();
template<class BinaryPredicate>
size_type unique(BinaryPredicate binary_pred);

void merge(list& x);
void merge(list&& x);
template<class Compare> void merge(list& x, Compare comp);
template<class Compare> void merge(list&& x, Compare comp);

void sort();
template<class Compare> void sort(Compare comp);
void reverse() noexcept;

};

template<class InputIterator, class Allocator = allocator<iter_value_type<InputIterator>>>
list<InputIterator, InputIterator, Allocator = Allocator>()
-> list<iter_value_type<InputIterator>, Allocator>;
template<class T, class Allocator>
    void swap(list<T, Allocator>& x, list<T, Allocator>& y)
        noexcept(noexcept(x.swap(y)));
}

An incomplete type \( T \) may be used when instantiating \( \text{list} \) if the allocator meets the allocator completeness requirements (16.4.4.6.2). \( T \) shall be complete before any member of the resulting specialization of \( \text{list} \) is referenced.

22.3.10.2 Constructors, copy, and assignment

explicit list(const Allocator&);
1 Effects: Constructs an empty list, using the specified allocator.
2 Complexity: Constant.

explicit list(size_type n, const Allocator& = Allocator());
3 Preconditions: \( T \) is \textit{Cpp17DefaultInsertable} into \*\this.  
4 Effects: Constructs a \textit{list} with \( n \) default-inserted elements using the specified allocator.
5 Complexity: Linear in \( n \).

list(size_type n, const T& value, const Allocator& = Allocator());
6 Preconditions: \( T \) is \textit{Cpp17CopyInsertable} into \*\this.
7 Effects: Constructs a \textit{list} with \( n \) copies of \textit{value}, using the specified allocator.
8 Complexity: Linear in \( n \).

template<class InputIterator>
    list(InputIterator first, InputIterator last, const Allocator& = Allocator());
9 Effects: Constructs a \textit{list} equal to the range \([\text{first}, \text{last})\).
10 Complexity: Linear in \text{distance}(\text{first}, \text{last}).

22.3.10.3 Capacity

void resize(size_type sz);
1 Preconditions: \( T \) is \textit{Cpp17DefaultInsertable} into \*\this.
2 Effects: If \text{size()} < \( sz \), appends \( sz - \text{size()} \) default-inserted elements to the sequence. If \( sz \leq \text{size()} \), equivalent to:

\[
\text{list<T>::iterator } \text{it} = \text{begin();}
\text{advance(it, sz);} \\
\text{erase(it, end());}
\]

void resize(size_type sz, const T& c);
3 Preconditions: \( T \) is \textit{Cpp17CopyInsertable} into \*\this.
4 Effects: As if by:

\[
\text{if (sz > size())}
    \text{insert(end(), sz-size(), c);} \\
\text{else if (sz < size())}
    \text{iterator i = begin();}
    \text{advance(i, sz);} \\
    \text{erase(i, end());}
\]
5 else
6    // do nothing

22.3.10.4 Modifiers

iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);

§ 22.3.10.4 858
template<class InputIterator>
iterator insert(const_iterator position, InputIterator first,
               InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);
template<class... Args> reference emplace_front(Args&&... args);
template<class... Args> reference emplace_back(Args&&... args);
template<class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);

Complexity: Insertion of a single element into a list takes constant time and exactly one call to a
constructor of T. Insertion of multiple elements into a list is linear in the number of elements inserted,
and the number of calls to the copy constructor or move constructor of T is exactly equal to the number
of elements inserted.

Remarks: Does not affect the validity of iterators and references. If an exception is thrown there are no
effects.

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_front();
void pop_back();
void clear() noexcept;

Effects: Invalidates only the iterators and references to the erased elements.
Throws: Nothing.
Complexity: Erasing a single element is a constant time operation with a single call to the destructor
of T. Erasing a range in a list is linear time in the size of the range and the number of calls to the
destructor of type T is exactly equal to the size of the range.

22.3.10.5 Operations

Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically
for them. In this subclause, arguments for a template parameter named Predicate or BinaryPredicate
shall meet the corresponding requirements in 25.2. For merge and sort, the definitions and requirements in
25.8 apply.

list provides three splice operations that destructively move elements from one list to another. The behavior
of splice operations is undefined if get_allocator() != x.get_allocator().

void splice(const_iterator position, list& x);
void splice(const_iterator position, list&& x);

Preconditions: addressof(x) != this && this is true.
Effects: Inserts the contents of x before position and x becomes empty. Pointers and references to the
moved elements of x now refer to those same elements but as members of *this. Iterators referring
to the moved elements will continue to refer to their elements, but they now behave as iterators into
*this, not into x.
Throws: Nothing.
Complexity: Constant time.

void splice(const_iterator position, list& x, const_iterator i);
void splice(const_iterator position, list&& x, const_iterator i);

Preconditions: i is a valid dereferenceable iterator of x.
Effects: Inserts an element pointed to by i from list x before position and removes the element from
x. The result is unchanged if position == i or position == ++i. Pointers and references to *i

232) As specified in 16.4.4.6, the requirements in this Clause apply only to lists whose allocators compare equal.
continue to refer to this same element but as a member of *this. Iterators to *i (including i itself) continue to refer to the same element, but now behave as iterators into *this, not into x.

**Throws:** Nothing.

**Complexity:** Constant time.

```cpp
void splice(const_iterator position, list& x, const_iterator first, 
const_iterator last);
void splice(const_iterator position, list&& x, const_iterator first, 
const_iterator last);
```

**Preconditions:** [first, last) is a valid range in x. position is not an iterator in the range [first, last).

**Effects:** Inserts elements in the range [first, last) before position and removes the elements from x. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

**Throws:** Nothing.

**Complexity:** Constant time.

```cpp
size_type remove(const T& value);
template<class Predicate> size_type remove_if(Predicate pred);
```

**Effects:** Erases all the elements in the list referred to by a list iterator i for which the following conditions hold: *i == value, pred(*i) != false. Invalidates only the iterators and references to the erased elements.

**Returns:** The number of elements erased.

**Throws:** Nothing unless an exception is thrown by *i == value or pred(*i) != false.

**Complexity:** Exactly size() applications of the corresponding predicate.

**Remarks:** Stable (16.4.6.8).

```cpp
size_type unique();
template<class BinaryPredicate> size_type unique(BinaryPredicate binary_pred);
```

**Effects:** Erases all but the first element from every consecutive group of equal elements referred to by the iterator i in the range [first + 1, last) for which *i == *(i-1) (for the version of unique with no arguments) or pred(*i, *(i - 1)) (for the version of unique with a predicate argument) holds. Invalidates only the iterators and references to the erased elements.

**Returns:** The number of elements erased.

**Throws:** Nothing unless an exception is thrown by *i == *(i-1) or pred(*i, *(i - 1)).

**Complexity:** If the range [first, last) is not empty, exactly (last - first) - 1 applications of the corresponding predicate, otherwise no applications of the predicate.

```cpp
void merge(list& x);
void merge(list&& x);
template<class Compare> void merge(list& x, Compare comp);
template<class Compare> void merge(list&& x, Compare comp);
```

**Preconditions:** Both the list and the argument list shall be sorted with respect to the comparator operator< (for the first two overloads) or comp (for the last two overloads), and get_allocator() == x.get_allocator() is true.

**Effects:** If addressof(x) == this, does nothing; otherwise, merges the two sorted ranges [begin(), end()) and (x.begin(), x.end()). The result is a range in which the elements will be sorted in non-decreasing order according to the ordering defined by comp; that is, for every iterator i, in the range other than the first, the condition comp(*i, *(i - 1)) will be false. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.
Complexity: At most size() + x.size() - 1 applications of comp if addressof(x) != this; otherwise, no applications of comp are performed. If an exception is thrown other than by a comparison there are no effects.

Remarks: Stable (16.4.6.8). If addressof(x) != this, the range [x.begin(), x.end()) is empty after the merge. No elements are copied by this operation.

void reverse() noexcept;

Effects: Reverses the order of the elements in the list. Does not affect the validity of iterators and references.

Complexity: Linear time.

void sort();

template<class Compare> void sort(Compare comp);

Effects: Sorts the list according to the operator< or a Compare function object. If an exception is thrown, the order of the elements in *this is unspecified. Does not affect the validity of iterators and references.

Complexity: Approximately N log N comparisons, where N == size().

Remarks: Stable (16.4.6.8).

22.3.10.6 Erasure

template<class T, class Allocator, class U>
typename list<T, Allocator>::size_type
erase(list<T, Allocator>& c, const U& value);

Effects: Equivalent to: return erase_if(c, [&](auto& elem) { return elem == value; });

template<class T, class Allocator, class Predicate>
typename list<T, Allocator>::size_type
erase_if(list<T, Allocator>& c, Predicate pred);

Effects: Equivalent to: return c.remove_if(pred);

22.3.11 Class template vector

22.3.11.1 Overview

A vector is a sequence container that supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency.

A vector meets all of the requirements of a container and of a reversible container (given in two tables in 22.2), of a sequence container, including most of the optional sequence container requirements (22.2.3), of an allocator-aware container (Table 76), and, for an element type other than bool, of a contiguous container (22.2.1). The exceptions are the push_front, pop_front, and emplace_front member functions, which are not provided. Descriptions are provided here only for operations on vector that are not described in one of these tables or for operations where there is additional semantic information.

The types iterator and const_iterator meet the constexpr iterator requirements (23.3.1).

namespace std {
    template<class T, class Allocator = allocator<T>>
    class vector {
        public:
            // types
            using value_type = T;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation_defined; // see 22.2
            using difference_type = implementation_defined; // see 22.2
            using iterator = implementation_defined; // see 22.2
            using const_iterator = implementation_defined; // see 22.2

§ 22.3.11.1 861
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;

// 22.3.11.2, construct/copy/destroy
constexpr vector() noexcept(noexcept(Allocator())) : vector(Allocator()) { }
constexpr explicit vector(const Allocator&) noexcept;
constexpr explicit vector(size_type n, const Allocator& = Allocator());
constexpr vector(initializer_list<T>, const Allocator& = Allocator());
template<class InputIterator>
  constexpr vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
constexpr vector(const vector& x);
constexpr vector(vector&&) noexcept(noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value || allocator_traits<Allocator>::is_always_equal::value));
template<class InputIterator>
  constexpr vector(const vector& x, vector&& x);
constexpr vector(const vector&, const Allocator&);
constexpr vector(vector&&, const Allocator&);
constexpr vector(initializer_list<T>, const Allocator& = Allocator());
constexpr ~vector();
constexpr vector& operator=(const vector& x);
constexpr vector& operator=(vector&& x)
  noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value || allocator_traits<Allocator>::is_always_equal::value);
constexpr vector& operator=(initializer_list<T>);

// iterators
constexpr iterator begin() noexcept;
constexpr const_iterator begin() const noexcept;
constexpr iterator end() noexcept;
constexpr const_iterator end() const noexcept;
constexpr reverse_iterator rbegin() noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator rend() noexcept;
constexpr const_reverse_iterator rend() const noexcept;
constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

// 22.3.11.3, capacity
[[nodiscard]] constexpr bool empty() const noexcept;
constexpr size_type size() const noexcept;
constexpr size_type max_size() const noexcept;
constexpr size_type capacity() const noexcept;
constexpr void resize(size_type sz);
constexpr void resize(size_type sz, const T& u);
constexpr void reserve(size_type n);
constexpr void shrink_to_fit();

// element access
constexpr reference operator[](size_type n);
constexpr const_reference operator[](size_type n) const;
constexpr const_reference at(size_type n) const;
constexpr reference at(size_type n);
constexpr reference front();
constexpr const_reference front() const;
constexpr reference back();
constexpr const_reference back() const;

// 22.3.11.4, data access
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;

// 22.3.11.5, modifiers
constexpr reference emplace_back(Args&&... args);
constexpr void push_back(const T& x);
constexpr void push_back(T&& x);
constexpr void pop_back();

template<class... Args> constexpr iterator emplace(const_iterator position, Args&&... args);
constexpr iterator insert(const_iterator position, const T& x);
constexpr iterator insert(const_iterator position, T&& x);
constexpr iterator insert(const_iterator position, size_type n, const T& x);

template<class... Args> constexpr iterator emplace(const_iterator position, InputIterator first, InputIterator last);
constexpr iterator insert(const_iterator position, initializer_list<T> il);

constexpr iterator erase(const_iterator position);
constexpr iterator erase(const_iterator first, const_iterator last);
constexpr void swap(vector&) noexcept;
constexpr void clear() noexcept;

// swap

template<class T, class Allocator>
constexpr void swap(vector<T, Allocator>& x, vector<T, Allocator>& y) noexcept(noexcept(x.swap(y)));

An incomplete type T may be used when instantiating vector if the allocator meets the allocator completeness requirements (16.4.4.6.2). T shall be complete before any member of the resulting specialization of vector is referenced.

22.3.11.2 Constructors

constexpr explicit vector(const Allocator&) noexcept;

Effects: Constructs an empty vector, using the specified allocator.

Complexity: Constant.

constexpr explicit vector(size_type n, const Allocator& = Allocator());

Preconditions: T is Cpp17DefaultInsertable into *this.

Effects: Constructs a vector with n default-inserted elements using the specified allocator.

Complexity: Linear in n.

constexpr vector(size_type n, const T& value,
const Allocator& = Allocator());

Preconditions: T is Cpp17CopyInsertable into *this.

Effects: Constructs a vector with n copies of value, using the specified allocator.

Complexity: Linear in n.

template<class InputIterator>
constexpr vector(InputIterator first, InputIterator last,
const Allocator& = Allocator());

Effects: Constructs a vector equal to the range [first, last), using the specified allocator.

§ 22.3.11.2
Complexity: Makes only \( N \) calls to the copy constructor of \( T \) (where \( N \) is the distance between \texttt{first} and \texttt{last}) and no reallocations if iterators \texttt{first} and \texttt{last} are of forward, bidirectional, or random access categories. It makes order \( N \) calls to the copy constructor of \( T \) and order \( \log N \) reallocations if they are just input iterators.

22.3.11.3 Capacity

\[
\text{constexpr size_type capacity() const noexcept;} \tag{vector.capacity}
\]

Returns: The total number of elements that the vector can hold without requiring reallocation.

Complexity: Constant time.

\[
\text{constexpr void reserve(size_type n);} \tag{reserve}
\]

Preconditions: \( T \) is \texttt{Cpp17MoveInsertable} into \*\texttt{this}.

Effects: A directive that informs a \texttt{vector} of a planned change in size, so that it can manage the storage allocation accordingly. After \texttt{reserve()}, \texttt{capacity()} is greater or equal to the argument of \texttt{reserve} if reallocation happens; and equal to the previous value of \texttt{capacity()} otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of \texttt{reserve}(). If an exception is thrown other than by the move constructor of a non-\texttt{Cpp17CopyInsertable} type, there are no effects.

Throws: \texttt{length_error} if \( n > \text{max\_size()} \).

Complexity: It does not change the size of the sequence and takes at most linear time in the size of the sequence.

Remarks: Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence, as well as the past-the-end iterator.

[Note 1: If no reallocation happens, they remain valid. — end note]

No reallocation shall take place during insertions that happen after a call to \texttt{reserve()} until an insertion would make the size of the vector greater than the value of \texttt{capacity()}.

\[
\text{constexpr void shrink\_to\_fit();} \tag{shrink_to_fit}
\]

Preconditions: \( T \) is \texttt{Cpp17MoveInsertable} into \*\texttt{this}.

Effects: \texttt{shrink\_to\_fit} is a non-binding request to reduce \texttt{capacity()} to \texttt{size()}.

[Note 2: The request is non-binding to allow latitude for implementation-specific optimizations. — end note]

It does not increase \texttt{capacity()}, but may reduce \texttt{capacity()} by causing reallocation. If an exception is thrown other than by the move constructor of a non-\texttt{Cpp17CopyInsertable} \( T \) there are no effects.

Complexity: If reallocation happens, linear in the size of the sequence.

Remarks: Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence as well as the past-the-end iterator.

[Note 3: If no reallocation happens, they remain valid. — end note]

\[
\text{constexpr void swap(vector& x);} \tag{swap}
\]

\[
\text{noexcept}(\text{allocator\_traits<Allocator>::propagate\_on\_container\_swap::value \|}
\]

\[\text{allocator\_traits<Allocator>::is\_always\_equal::value});\]

Effects: Exchanges the contents and \texttt{capacity()} of \*\texttt{this} with that of \texttt{x}.

Complexity: Constant time.

\[
\text{constexpr void resize(size_type sz);} \tag{resize}
\]

Preconditions: \( T \) is \texttt{Cpp17MoveInsertable} and \texttt{Cpp17DefaultInsertable} into \*\texttt{this}.

Effects: If \( sz < \text{size}() \), erases the last \texttt{size()} - \( sz \) elements from the sequence. Otherwise, appends \( sz - \text{size}() \) default-inserted elements to the sequence.

Remarks: If an exception is thrown other than by the move constructor of a non-\texttt{Cpp17CopyInsertable} \( T \) there are no effects.

---

233) \texttt{reserve()} uses \texttt{Allocator::allocates()} which can throw an appropriate exception.
constexpr void resize(size_type sz, const T& c);

17 Preconditions: T is Cpp17CopyInsertable into *this.
18 Effects: If sz < size(), erases the last size() - sz elements from the sequence. Otherwise, appends sz - size() copies of c to the sequence.
19 Remarks: If an exception is thrown there are no effects.

22.3.11.4 Data

constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
1 Returns: A pointer such that [data(), data() + size()) is a valid range. For a non-empty vector, data() == addressof(front()).
2 Complexity: Constant time.

22.3.11.5 Modifiers

constexpr iterator insert(const_iterator position, const T& x);
constexpr iterator insert(const_iterator position, T&& x);
constexpr iterator insert(const_iterator position, size_type n, const T& x);
template<class InputIterator>
constexpr iterator insert(const_iterator position, InputIterator first, InputIterator last);
constexpr iterator insert(const_iterator position, initializer_list<T>);
template<class... Args> constexpr reference emplace_back(Args&&... args);
template<class... Args> constexpr iterator emplace(const_iterator position, Args&&... args);
constexpr void push_back(const T& x);
constexpr void push_back(T&& x);
1 Complexity: If reallocation happens, linear in the number of elements of the resulting vector; otherwise, linear in the number of elements inserted plus the distance to the end of the vector.
2 Remarks: Causes reallocation if the new size is greater than the old capacity. Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence, as well as the past-the-end iterator. If no reallocation happens, then references, pointers, and iterators before the insertion point remain valid but those at or after the insertion point, including the past-the-end iterator, are invalidated. If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of T or by any InputIterator operation there are no effects. If an exception is thrown while inserting a single element at the end and T is Cpp17CopyInsertable or is_nothrow_move_constructible_v<T> is true, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-Cpp17CopyInsertable T, the effects are unspecified.

constexpr iterator erase(const_iterator position);
constexpr iterator erase(const_iterator first, const_iterator last);
constexpr void pop_back();
3 Effects: Invalidates iterators and references at or after the point of the erase.
4 Throws: Nothing unless an exception is thrown by the assignment operator or move assignment operator of T.
5 Complexity: The destructor of T is called the number of times equal to the number of the elements erased, but the assignment operator of T is called the number of times equal to the number of elements in the vector after the erased elements.

22.3.11.6 Erasure

template<class T, class Allocator, class U>
constexpr typename vector<T, Allocator>::size_type
erase(vector<T, Allocator>& c, const U& value);
1 Effects: Equivalent to:
   auto it = remove(c.begin(), c.end(), value);
   auto r = distance(it, c.end());
   c.erase(it, c.end());
return r;

// Function to erase elements if a predicate returns true

template<class T, class Allocator, class Predicate>
constexpr typename vector<T, Allocator>::size_type
erase_if(vector<T, Allocator>& c, Predicate pred);

Effects: Equivalent to:

auto it = remove_if(c.begin(), c.end(), pred);
auto r = distance(it, c.end());
c.erase(it, c.end());
return r;

22.3.12 Class vector<bool>

To optimize space allocation, a specialization of vector for bool elements is provided:

namespace std {
    template<class Allocator>
    class vector<bool, Allocator> {
public:
    // Types
    using value_type = bool;
    using allocator_type = Allocator;
    using pointer = implementation-defined;
    using const_pointer = implementation-defined;
    using const_reference = bool;
    using size_type = implementation-defined; // see 22.2
    using difference_type = implementation-defined; // see 22.2
    using iterator = implementation-defined; // see 22.2
    using const_iterator = implementation-defined; // see 22.2
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // Bit reference
    class reference {
        friend class vector;
        constexpr reference() noexcept;
        constexpr reference(const reference&) = default;
        constexpr ~reference();
        constexpr operator bool() const noexcept;
        constexpr reference& operator=(const bool x) noexcept;
        constexpr reference& operator=(const reference& x) noexcept;
        constexpr void flip() noexcept; // Flips the bit
    };

    // Construct/copy/destroy
    constexpr vector() : vector(Allocator()) { }
    constexpr explicit vector(const Allocator&);
    constexpr explicit vector(size_type n, const Allocator& = Allocator());
    constexpr vector(const allocator_type& value, const Allocator& = Allocator());
    template<class InputIterator>
    constexpr vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
    constexpr vector(const vector& x);
    constexpr vector(vector&& x);
    constexpr vector(const vector&, const Allocator&);
    constexpr vector(vector&, const Allocator&);
    constexpr vector(initializer_list<bool>, const Allocator& = Allocator());
    constexpr vector() noexcept;
    constexpr ~vector() noexcept;
    constexpr operator=(const vector& x);
    constexpr operator=(vector& x);
    constexpr vector& operator=(const bool x);
    constexpr vector& operator=(vector&& x);
    template<class InputIterator>
    constexpr void assign(InputIterator first, InputIterator last);
    constexpr void assign(size_type n, const bool& t);
}
constexpr void assign(initializer_list<bool>);
constexpr allocator_type get_allocator() const noexcept;

// iterators
constexpr iterator begin() noexcept;
constexpr const_iterator begin() const noexcept;
constexpr iterator end() noexcept;
constexpr const_iterator end() const noexcept;
constexpr reverse_iterator rbegin() noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator rend() noexcept;
constexpr const_reverse_iterator rend() const noexcept;

// capacity
[[nodiscard]] constexpr bool empty() const noexcept;
constexpr size_type size() const noexcept;
constexpr size_type max_size() const noexcept;
constexpr size_type capacity() const noexcept;
constexpr void resize(size_type sz, bool c = false);
constexpr void reserve(size_type n);
constexpr void shrink_to_fit();

// element access
constexpr reference operator[](size_type n);
constexpr const_reference operator[](size_type n) const;
constexpr const_reference at(size_type n) const;
constexpr reference at(size_type n);
constexpr reference front();
constexpr const_reference front() const;
constexpr reference back();
constexpr const_reference back() const;

// modifiers
template<class... Args> constexpr reference emplace_back(Args&&... args);
constexpr void push_back(const bool& x);
constexpr void pop_back();
template<class... Args> constexpr iterator emplace(const_iterator position, Args&&... args);
constexpr iterator insert(const_iterator position, const bool& x);
constexpr iterator insert(const_iterator position, size_type n, const bool& x);
template<class InputIterator>
constexpr iterator insert(const_iterator position, InputIterator first, InputIterator last);
constexpr iterator insert(const_iterator position, initializer_list<bool> il);

constexpr iterator erase(const_iterator position);
constexpr iterator erase(const_iterator first, const_iterator last);
constexpr void swap(vector&);
constexpr static void swap(reference x, reference y) noexcept;
constexpr void flip() noexcept; // flips all bits
constexpr void clear() noexcept;

2 Unless described below, all operations have the same requirements and semantics as the primary vector template, except that operations dealing with the bool value type map to bit values in the container storage and allocator_traits::construct (20.10.9.3) is not used to construct these values.

3 There is no requirement that the data be stored as a contiguous allocation of bool values. A space-optimized representation of bits is recommended instead.
reference is a class that simulates the behavior of references of a single bit in vector<bool>. The conversion function returns true when the bit is set, and false otherwise. The assignment operator sets the bit when the argument is (convertible to) true and clears it otherwise. flip reverses the state of the bit.

constexpr void flip() noexcept;

Effects: Replaces each element in the container with its complement.

constexpr static void swap(reference x, reference y) noexcept;

Effects: Exchanges the contents of x and y as if by:

```cpp
bool b = x;
x = y;
y = b;
```

template<class Allocator> struct hash<vector<bool, Allocator>>;

The specialization is enabled (20.14.19).

### 22.4 Associative containers

#### 22.4.1 In general

The header `<map>` defines the class templates map and multimap; the header `<set>` defines the class templates set and multiset.

The following exposition-only alias templates may appear in deduction guides for associative containers:

```cpp
template<class InputIterator>
using iter_value_type = 
typename iterator_traits<InputIterator>::value_type; // exposition only
template<class InputIterator>
using iter_key_type = remove_const_t<
typename iterator_traits<InputIterator>::value_type::first_type>;
// exposition only
template<class InputIterator>
using iter_mapped_type =
typename iterator_traits<InputIterator>::value_type::second_type;
// exposition only
template<class InputIterator>
using iter_to_alloc_type = pair<
add_const_t<typename iterator_traits<InputIterator>::value_type::first_type>,
typename iterator_traits<InputIterator>::value_type::second_type>;
// exposition only
```

#### 22.4.2 Header `<map>` synopsis

```cpp
#include <compare>  // see 17.11.1
#include <initializer_list>  // see 17.10.2

namespace std {
    // 22.4.4, class template map
    template<class Key, class T, class Compare = less<Key>,
             class Allocator = allocator<pair<const Key, T>>> class map;

    template<class Key, class T, class Compare, class Allocator>
    bool operator==(const map<Key, T, Compare, Allocator>& x,
                    const map<Key, T, Compare, Allocator>& y);

    template<class Key, class T, class Compare, class Allocator>
    synth-three-way-result<pair<const Key, T>>
    operator<=>(const map<Key, T, Compare, Allocator>& x,
                 const map<Key, T, Compare, Allocator>& y);

    template<class Key, class T, class Compare, class Allocator>
    void swap(map<Key, T, Compare, Allocator>& x,
              map<Key, T, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));
}
```
template<class Key, class T, class Compare, class Allocator, class Predicate>
  typename map<Key, T, Compare, Allocator>::size_type
  erase_if(map<Key, T, Compare, Allocator>& c, Predicate pred);

// 22.4.5, class template multimap
template<class Key, class T, class Compare = less<Key>,
  class Allocator = allocator<pair<const Key, T>>>
  class multimap;

template<class Key, class T, class Compare, class Allocator>
  bool operator==(const multimap<Key, T, Compare, Allocator>& x,
      const multimap<Key, T, Compare, Allocator>& y);

template<class Key, class T, class Compare, class Allocator>
  synth-three-way-result<
      pair<const Key, T>>
      operator<=>(const multimap<Key, T, Compare, Allocator>& x,
          const multimap<Key, T, Compare, Allocator>& y);

// 22.4.3 Header <set> synopsis

namespace std {
  // 22.4.6, class template set
  template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
    class set;

  template<class Key, class Compare, class Allocator>
    bool operator==(const set<Key, Compare, Allocator>& x,
        const set<Key, Compare, Allocator>& y);

  template<class Key, class Compare, class Allocator>
    synth-three-way-result<Key>
    operator<=>(const set<Key, Compare, Allocator>& x,
        const set<Key, Compare, Allocator>& y);

  template<class Key, class Compare, class Allocator>
    void swap(set<Key, Compare, Allocator>& x,
        set<Key, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));

  template<class Key, class Compare, class Allocator, class Predicate>
    typename set<Key, Compare, Allocator>::size_type
    erase_if(set<Key, Compare, Allocator>& c, Predicate pred);
}

§ 22.4.3 869
22.4.7, class template multiset

// 22.4.7, class template multiset
template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
class multiset;

template<class Key, class Compare, class Allocator>
bool operator==(const multiset<Key, Compare, Allocator>& x,
              const multiset<Key, Compare, Allocator>& y);

template<class Key, class Compare, class Allocator>
synth-three-way-result<Key> operator<=>(const multiset<Key, Compare, Allocator>& x,
                                        const multiset<Key, Compare, Allocator>& y);

template<class Key, class Compare, class Allocator>
void swap(multiset<Key, Compare, Allocator>& x,
          multiset<Key, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));

template<class Key, class Compare, class Allocator, class Predicate>
type::multiset<Key, Compare, Allocator>::size_type
erase_if(multiset<Key, Compare, Allocator>& c,
         Predicate pred);

namespace pmr {
    template<class Key, class Compare = less<Key>>
    using set = std::set<Key, Compare, polymorphic_allocator<Key>>;
    template<class Key, class Compare = less<Key>>
    using multiset = std::multiset<Key, Compare, polymorphic_allocator<Key>>;
}

namespace std {
    template<class Key, class T, class Compare = less<Key>,
             class Allocator = allocator<pair<const Key, T>>>
    class map {
        public:
        // types
        using key_type = Key;
        using mapped_type = T;
        using value_type = pair<const Key, T>;
        using key_compare = Compare;
        using allocator_type = Allocator;
        using pointer = typename allocator_traits<Allocator>::pointer;
        using const_pointer = typename allocator_traits<Allocator>::const_pointer;
        using reference = value_type&;
        using const_reference = const value_type&;
        using size_type = implementation-defined; // see 22.2
        using difference_type = implementation-defined; // see 22.2
        using iterator = implementation-defined; // see 22.2
        using const_iterator = implementation-defined; // see 22.2
        using reverse_iterator = std::reverse_iterator<iterator>;
        using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    };

§ 22.4.4.1
using node_type = unspecified;
using insert_return_type = insert-return-type<iterator, node_type>;

class value_compare {
friend class map;
protected:
  Compare comp;
value_compare(Compare c) : comp(c) {}
public:
  bool operator()(const value_type& x, const value_type& y) const {
    return comp(x.first, y.first);
  }
};

// 22.4.4.2, construct/copy/destroy
map() : map(Compare()) { }
explicit map(Compare comp, const Allocator& = Allocator());
template<class InputIterator>
map(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& = Allocator());
map(const map& x);
map(map& x);
explicit map(const Allocator&);
map(const map&, const Allocator&);
map(map&, const Allocator&);
map(initializer_list<value_type>,
    const Compare& = Compare(),
    const Allocator& = Allocator());
template<class InputIterator>
map(InputIterator first, InputIterator last, const Allocator& a) :
    map(first, last, Compare(), a) { }
map(initializer_list<value_type> il, const Allocator& a) :
    map(il, Compare(), a) { }
~map();
map& operator=(const map& x);
map& operator=(map& x)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
            is_nothrow_move_assignable_v<Compare>);
map& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 22.4.4.3, element access
mapped_type& operator[](const key_type& x);
mapped_type& operator[](key_type&& x);
const mapped_type& at(const key_type& x);
const mapped_type& at(const key_type& x) const;

// 22.4.4.4, modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
template<class P> pair<iterator, bool> insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class P>
    iterator insert(const_iterator position, P&&);
template<class InputIterator>
    void insert(InputIterator first, InputIterator last);
    void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

    template<class... Args>
        pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
    template<class... Args>
        pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
    template<class... Args>
        iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
    template<class... Args>
        iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
    template<class M>
        pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
    template<class M>
        pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
    template<class M>
        iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
    template<class M>
        iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);

void swap(map&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
    is_nothrow_swappable_v<Compare>);

    template<class C2>
        void merge(map<Key, T, C2, Allocator>& source);
    template<class C2>
        void merge(map<Key, T, C2, Allocator>&& source);
    template<class C2>
        void merge(multimap<Key, T, C2, Allocator>& source);
    template<class C2>
        void merge(multimap<Key, T, C2, Allocator>&& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// map operations
iterator         find(const key_type& x);
const_iterator find(const key_type& x) const;
template<class K> iterator find(const K& x);
template<class K> const_iterator find(const K& x) const;
size_type count(const key_type& x) const;
template<class K> size_type count(const K& x) const;
bool contains(const key_type& x) const;
template<class K> bool contains(const K& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
template<class K> iterator lower_bound(const K& x);
template<class K> const_iterator lower_bound(const K& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
template<class K> iterator upper_bound(const K& x);
template<class K> const_iterator upper_bound(const K& x) const;
pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
template<class K>
pair<iterator, iterator> equal_range(const K& x);
template<class K>
pair<const_iterator, const_iterator> equal_range(const K& x) const;
};

template<class InputIterator, class Compare = less<
iter-key-type<InputIterator>>,
class Allocator = allocator<
iter-to-alloc-type<InputIterator>>> map(InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator())
-> map<
iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Compare, Allocator>;
template<class Key, class T, class Compare = less<Key>,
class Allocator = allocator<pair<const Key, T>>>
map(initializer_list<pair<Key, T>>, Compare = Compare(), Allocator = Allocator())
-> map<Key, T, Compare, Allocator>;
template<class InputIterator, class Allocator>
map(InputIterator, InputIterator, Allocator)
-> map<
iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
less<
iter-key-type<InputIterator>>, Allocator>;
template<class Key, class T, class Allocator>
map(initializer_list<pair<Key, T>>, Allocator) -> map<Key, T, less<Key>, Allocator>;

// swap
template<class Key, class T, class Compare, class Allocator>
void swap(map<Key, T, Compare, Allocator>& x,
    map<Key, T, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));
}

22.4.4.2 Constructors, copy, and assignment [map.cons]
explicit map(const Compare& comp, const Allocator& = Allocator());

Effects: Constructs an empty map using the specified comparison object and allocator.

Complexity: Constant.

template<class InputIterator>
map(InputIterator first, InputIterator last,
const Compare& comp = Compare(), const Allocator& = Allocator());

Effects: Constructs an empty map using the specified comparison object and allocator, and inserts elements from the range [first, last).

Complexity: Linear in \(N\) if the range [first, last) is already sorted using \(\text{comp}\) and otherwise \(N \log N\), where \(N\) is last - first.

### 22.4.4.3 Element access

mapped_type& operator[](const key_type& x);

Effects: Equivalent to: return try_emplace(x).first->second;

mapped_type& operator[](key_type&& x);

Effects: Equivalent to: return try_emplace(move(x)).first->second;

mapped_type& at(const key_type& x);
const mapped_type& at(const key_type& x) const;

Returns: A reference to the mapped_type corresponding to \(x\) in *this.

Throws: An exception object of type out_of_range if no such element is present.

Complexity: Logarithmic.

### 22.4.4.4 Modifiers

template<class P>
pair<iterator, bool> insert(P&& x);

template<class P>
iterator insert(const_iterator position, P&& x);

Constraints: is_constructible_v<value_type, P&&> is true.

Effects: The first form is equivalent to return emplace(std::forward<P>(x)). The second form is equivalent to return emplace_hint(position, std::forward<P>(x)).

template<class... Args>
pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);

template<class... Args>
iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);

Preconditions: value_type is Cpp17EmplaceConstructible into map from piecewise_construct, forward_as_tuple(k), forward_as_tuple(std::forward<Args>(args)...).

Effects: If the map already contains an element whose key is equivalent to \(k\), there is no effect. Otherwise inserts an object of type value_type constructed with piecewise_construct, forward_as_tuple(k), forward_as_tuple(std::forward<Args>(args)...).

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to \(k\).

Complexity: The same as emplace and emplace_hint, respectively.

template<class... Args>
pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);

template<class... Args>
iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);

Preconditions: value_type is Cpp17EmplaceConstructible into map from piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...).

Effects: If the map already contains an element whose key is equivalent to \(k\), there is no effect. Otherwise inserts an object of type value_type constructed with piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...).

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to \(k\).

Complexity: The same as emplace and emplace_hint, respectively.
template<class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);

template<class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);

Mandates: is_assignable_v<mapped_type&, M&&> is true.

Preconditions: value_type is Cpp17EmplaceConstructible into map from k, forward<M>(obj).

Effects: If the map already contains an element e whose key is equivalent to k, assigns std::forward<M>(obj) to e.second. Otherwise inserts an object of type value_type constructed with k, std::forward<M>(obj).

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

template<class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);

template<class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

Mandates: is_assignable_v<mapped_type&, M&&> is true.

Preconditions: value_type is Cpp17EmplaceConstructible into map from move(k), forward<M>(obj).

Effects: If the map already contains an element e whose key is equivalent to k, assigns std::forward<M>(obj) to e.second. Otherwise inserts an object of type value_type constructed with std::move(k), std::forward<M>(obj).

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

22.4.4.5 Erasure

template<class Key, class T, class Compare, class Allocator, class Predicate>
  typename multimap<Key, T, Compare, Allocator>::size_type
  erase_if(map<Key, T, Compare, Allocator>& c, Predicate pred);

Effects: Equivalent to:

```cpp
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}
return original_size - c.size();
```

22.4.5 Class template multimap

22.4.5.1 Overview

A multimap is an associative container that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type T based on the keys. The multimap class supports bidirectional iterators.

A multimap meets all of the requirements of a container and of a reversible container (22.2), of an associative container (22.2.6), and of an allocator-aware container (Table 76). A multimap also provides most operations described in 22.2.6 for equal keys. This means that a multimap supports the a_eq operations in 22.2.6 but not the a_uniq operations. For a multimap<Key,T> the key_type is Key and the value_type is pair<const Key,T>. Descriptions are provided here only for operations on multimap that are not described in one of those tables or for operations where there is additional semantic information.
namespace std {

    template<class Key, class T, class Compare = less<Key>,
             class Allocator = allocator<pair<const Key, T>>>
    class multimap {
        public:
            // types
            using key_type = Key;
            using mapped_type = T;
            using value_type = pair<const Key, T>;
            using key_compare = Compare;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation_defined;
            // see 22.2
            using difference_type = implementation_defined;
            // see 22.2
            using iterator = implementation_defined;
            // see 22.2
            using const_iterator = implementation_defined;
            // see 22.2
            using reverse_iterator = std::reverse_iterator<iterator>;
            using const_reverse_iterator = std::reverse_iterator<const_iterator>;
            using node_type = unspecified;

            class value_compare {
                friend class multimap;
            protected:
                Compare comp;
            value_compare(Compare c) : comp(c) {} ;
        public:
            bool operator()(const value_type& x, const value_type& y) const {
                return comp(x.first, y.first);
            }
        };

        // § 22.4.5.2, construct/copy/destroy
        multimap() : multimap(Compare()) {} ;
        explicit multimap(const Compare& comp, const Allocator& = Allocator());
        template<class InputIterator>
        multimap(InputIterator first, InputIterator last,
                  const Compare& comp = Compare(),
                  const Allocator& = Allocator());
        multimap(const multimap& x);
        multimap(multimap&& x);
        explicit multimap(const Allocator&);
        multimap(multimap&, const Allocator&);
        multimap(initializer_list<value_type>,
                 const Compare& = Compare(),
                 const Allocator& = Allocator());
        template<class InputIterator>
        multimap(InputIterator first, InputIterator last, const Allocator& a)
        : multimap(first, last, Compare(), a) {} ;
        multimap(initializer_list<value_type> il, const Allocator& a)
        : multimap(il, Compare(), a) {} ;
        ~multimap();
        multimap& operator=(const multimap& x);
        multimap& operator=(multimap&& x)
        noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                  is_nothrow_move_assignable_v<Compare>);
        multimap& operator=(initializer_list<value_type>);
        allocator_type get_allocator() const noexcept;

        // iterators
        iterator begin() noexcept;
    };

§ 22.4.5.1 876
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 22.4.5.3, modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
template<class P> iterator insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class P> iterator insert(const_iterator position, P&& x);
template<class InputIterator>
void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator erase(const_iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
size_type erase(const_iterator first, const_iterator last);
void swap(multimap&) noexcept(allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_swappable_v<Compare>);
void clear() noexcept;

template<class C2>
void merge(multimap<Key, T, C2, Allocator>& source);
template<class C2>
void merge(multimap<Key, T, C2, Allocator>&& source);
template<class C2>
void merge(map<Key, T, C2, Allocator>& source);
template<class C2>
void merge(map<Key, T, C2, Allocator>&& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// map operations
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
template<class K> iterator find(const K& x);
template<class K> const_iterator find(const K& x) const;
size_type count(const key_type& x) const;
template<class K> size_type count(const K& x) const;

bool contains(const key_type& x) const;
template<class K> bool contains(const K& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
template<class K> iterator lower_bound(const K& x);
template<class K> const_iterator lower_bound(const K& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
template<class K> iterator upper_bound(const K& x);
template<class K> const_iterator upper_bound(const K& x) const;

pair<iterator, iterator> equal_range(const key_type& x);
const_iterator equal_range(const key_type& x) const;
template<class K> pair<iterator, iterator> equal_range(const K& x);
template<class K> const_iterator equal_range(const K& x) const;


 template<class InputIterator, class Compare = less<
    iter-key-type<InputIterator>>,
    class Allocator = allocator<
    iter-to-alloc-type<InputIterator>>>
multimap(InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator())
-> multimap<
    iter-key-type<InputIterator>,
    iter-mapped-type<InputIterator>,
    Compare, Allocator>;

template<class Key, class T, class Compare = less<Key>,
    class Allocator = allocator<pair<const Key, T>>>
multimap(initializer_list<pair<Key, T>>, Compare = Compare(), Allocator = Allocator())
-> multimap<
    Key, T, Compare, Allocator>;

template<class InputIterator, class Allocator>
multimap(InputIterator, InputIterator, Allocator)
-> multimap<
    iter-key-type<InputIterator>,
    iter-mapped-type<InputIterator>,
    less<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
multimap(initializer_list<pair<Key, T>>, Allocator)
-> multimap<
    Key, T, less<Key>, Allocator>;

// swap
template<class Key, class T, class Compare, class Allocator>
void swap(multimap<Key, T, Compare, Allocator>& x,
          multimap<Key, T, Compare, Allocator>& y)
  noexcept(noexcept(x.swap(y)));
}

22.4.5.2 Constructors

explicit multimap(const Compare& comp, const Allocator& = Allocator());

Effects: Constructs an empty multimap using the specified comparison object and allocator.

Complexity: Constant.

template<class InputIterator>
multimap(InputIterator first, InputIterator last,
         const Compare& comp = Compare(),
         const Allocator& = Allocator());

Effects: Constructs an empty multimap using the specified comparison object and allocator, and inserts
elements from the range [first, last).
Complexity: Linear in \( N \) if the range \([\text{first}, \text{last})\) is already sorted using \text{comp} and otherwise \( N \log N \), where \( N = \text{last} - \text{first} \).

22.4.5.3 Modifiers

\texttt{template<class P> iterator insert(P&& x);} \\
\texttt{template<class P> iterator insert(const_iterator position, P&& x);} \\

Constraints: is\_constructible\_v\langle value\_type, P&&\rangle \text{ is true}.

Effects: The first form is equivalent to return \texttt{emplace(std::forward<P>(x))}. The second form is equivalent to return \texttt{emplace\_hint(position, std::forward<P>(x))}.

22.4.5.4 Erasure

\texttt{template<class Key, class T, class Compare, class Allocator, class Predicate> \\
\texttt{typename multimap\langle Key, T, Compare, Allocator\rangle::size\_type \\
\texttt{erase\_if(multimap\langle Key, T, Compare, Allocator\& c, Predicate pred);}} \\

Effects: Equivalent to:

```cpp
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}
return original_size - c.size();
```

22.4.6 Class template \texttt{set}

22.4.6.1 Overview

A \texttt{set} is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves. The \texttt{set} class supports bidirectional iterators.

A \texttt{set} meets all of the requirements of a container, of a reversible container (22.2), of an associative container (22.2.6), and of an allocator-aware container (Table 76). A \texttt{set} also provides most operations described in 22.2.6 for unique keys. This means that a \texttt{set} supports the \texttt{a\_uniq} operations in 22.2.6 but not the \texttt{a\_eq} operations. For a \texttt{set\langle Key\rangle} both the \texttt{key\_type} and \texttt{value\_type} are \texttt{Key}. Descriptions are provided here only for operations on \texttt{set} that are not described in one of these tables and for operations where there is additional semantic information.

```cpp
namespace std {
  template<class Key, class Compare = less<Key>, 
  class Allocator = allocator<Key>>
  class set {
  
  public:
  // types
  using key\_type = Key;
  using key\_compare = Compare;
  using value\_type = Key;
  using value\_compare = Compare;
  using allocator\_type = Allocator;
  using pointer = typename allocator\_traits<Allocator>::pointer;
  using const\_pointer = typename allocator\_traits<Allocator>::const\_pointer;
  using reference = value\_type&;
  using const\_reference = const value\_type&;
  using size\_type = implementation\_defined; // see 22.2
  using difference\_type = implementation\_defined; // see 22.2
  using iterator = implementation\_defined; // see 22.2
  using const\_iterator = implementation\_defined; // see 22.2
  using reverse\_iterator = std::reverse\_iterator<iterator>;
  using const\_reverse\_iterator = std::reverse\_iterator<const\_iterator>;
  using node\_type = unspecified;
  using insert\_return\_type = insert\_return\_type<iterator, node\_type>;
```
// 22.4.6.2, construct/copy/destroy
set() : set(Compare()) { }
explicit set(const Compare& comp, const Allocator& = Allocator());
template<class InputIterator>
set(InputIterator first, InputIterator last,
const Compare& comp = Compare(), const Allocator& = Allocator());
set(const set& x);
set(set&& x);
explicit set(const Allocator&);
set(const set&, const Allocator&);
set(set& const Allocator&);
set(initializer_list<value_type>, const Compare& comp = Compare(),
const Allocator& = Allocator());
template<class InputIterator>
set(InputIterator first, InputIterator last, const Allocator& a)
: set(first, last, Compare(), a) { }
set(initializer_list<value_type> il, const Allocator& a)
: set(il, Compare(), a) { }
~set();
set& operator=(const set& x);
set& operator=(set&& x)
   noexcept(algorithm::traits<Allocator>::is_always_equal::value &&
   noexcept_traits_value<typename Compare>::nothrow_move_assignable_v);
set& operator=(initializer_list<value_type>);*
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);*
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class InputIterator>
   void insert(InputIterator first, InputIterator last);
   void insert(initializer_list<value_type>);
node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type& nh);
iterator insert(const_iterator hint, node_type& nh);
iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(set&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
    is_nothrow_swappable_v<Compare>);
void clear() noexcept;

template<class C2>
  void merge(set<Key, C2, Allocator>& source);
template<class C2>
  void merge(set<Key, C2, Allocator>&& source);
template<class C2>
  void merge(multiset<Key, C2, Allocator>& source);
template<class C2>
  void merge(multiset<Key, C2, Allocator>&& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// set operations
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
template<class K> iterator find(const K& x);
template<class K> const_iterator find(const K& x) const;
size_type count(const key_type& x) const;
template<class K> size_type count(const K& x) const;
bool contains(const key_type& x) const;
template<class K> bool contains(const K& x) const;
iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
template<class K> iterator lower_bound(const K& x);
template<class K> const_iterator lower_bound(const K& x) const;
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
template<class K> iterator upper_bound(const K& x);
template<class K> const_iterator upper_bound(const K& x) const;
pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
template<class K>
  pair<iterator, iterator> equal_range(const K& x);
template<class K>
  pair<const_iterator, const_iterator> equal_range(const K& x) const;
};

template<class InputIterator,
  class Compare = less<iter_value_type<InputIterator>>,
  class Allocator = allocator<iter_value_type<InputIterator>>>
set(InputIterator, InputIterator,
  Compare = Compare(), Allocator = Allocator())
  -> set<iter_value_type<InputIterator>, Compare, Allocator>;

template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
set(initializer_list<Key>, Compare = Compare(), Allocator = Allocator())
  -> set<Key, Compare, Allocator>;

§ 22.4.6.1
template<class InputIterator, class Allocator>
  set(InputIterator, InputIterator, Allocator)
  -> set<typename iter_value_type<InputIterator>::less<input_value_type<InputIterator>>, Allocator>;

template<class Key, class Allocator>
  set(initializer_list<Key>, Allocator) -> set<Key, less<Key>, Allocator>;

// swap
template<class Key, class Compare, class Allocator>
  void swap(set<Key, Compare, Allocator>& x, set<Key, Compare, Allocator>& y)
  noexcept(noexcept(x.swap(y)));

22.4.6.2 Constructors, copy, and assignment

explicit set(const Compare& comp, const Allocator& = Allocator());
1     Effects: Constructs an empty set using the specified comparison objects and allocator.
2     Complexity: Constant.

template<class InputIterator>
  set(InputIterator first, InputIterator last, const Compare& comp = Compare(), const Allocator& = Allocator());
3     Effects: Constructs an empty set using the specified comparison object and allocator, and inserts elements from the range [first, last).
4     Complexity: Linear in N if the range [first, last) is already sorted using comp and otherwise N log N, where N is last - first.

22.4.6.3 Erasure

template<class Key, class Compare, class Allocator, class Predicate>
  typename set<Key, Compare, Allocator>::size_type
  erase_if(set<Key, Compare, Allocator>& c, Predicate pred);
1     Effects: Equivalent to:
             auto original_size = c.size();
             for (auto i = c.begin(), last = c.end(); i != last; ) {
                 if (pred(*i)) {
                     i = c.erase(i);
                 } else {
                     ++i;
                 }
             }
             return original_size - c.size();

22.4.7 Class template multiset

22.4.7.1 Overview
1     A multiset is an associative container that supports equivalent keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves. The multiset class supports bidirectional iterators.

2     A multiset meets all of the requirements of a container, of a reversible container (22.2), of an associative container (22.2.6), and of an allocator-aware container (Table 76). multiset also provides most operations described in 22.2.6 for duplicate keys. This means that a multiset supports the a_eq operations in 22.2.6 but not the a_uniq operations. For a multiset<Key> both the key_type and value_type are Key. Descriptions are provided here only for operations on multiset that are not described in one of these tables and for operations where there is additional semantic information.
namespace std {
  template<class Key, class Compare = less<Key>,
            class Allocator = allocator<Key>>
  class multiset {
  public:
    // types
    using key_type = Key;
    using key_compare = Compare;
    using value_type = Key;
    using value_compare = Compare;
    using allocator_type = Allocator;
    using pointer = typename allocator_traits<Allocator>::pointer;
    using const_pointer = typename allocator_traits<Allocator>::const_pointer;
    using reference = value_type&;
    using const_reference = const value_type&;
    using size_type = implementation-defined; // see 22.2
    using difference_type = implementation-defined; // see 22.2
    using iterator = implementation-defined; // see 22.2
    using const_iterator = implementation-defined; // see 22.2
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    using node_type = unspecified;

    // 22.4.7.2, construct/copy/destroy
    multiset() : multiset(Compare()) { }
    explicit multiset(const Compare& comp, const Allocator& = Allocator());
    template<class InputIterator>
    multiset(InputIterator first, InputIterator last,
             const Compare& comp = Compare(), const Allocator& = Allocator());
    multiset(const multiset& x);
    multiset(multiset&& x);
    explicit multiset(const Allocator&);
    multiset(const multiset&, const Allocator&);
    multiset(initializer_list<value_type>, const Compare& = Compare(),
             const Allocator& = Allocator());
    template<class InputIterator>
    multiset(InputIterator first, InputIterator last, const Allocator& a)
             : multiset(first, last, Compare(), a) { }
    multiset(initializer_list<value_type> il, const Allocator& a)
             : multiset(il, Compare(), a) { }
    ~multiset();
    multiset& operator=(const multiset& x);
    multiset& operator=(multiset&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                is_nothrow_move_assignable_v<Compare>);
    multiset& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

    // iterators
    iterator begin() noexcept;
    const_iterator begin() const noexcept;
    iterator end() noexcept;
    const_iterator end() const noexcept;
    reverse_iterator rbegin() noexcept;
    const_reverse_iterator rbegin() const noexcept;
    reverse_iterator rend() noexcept;
    const_reverse_iterator rend() const noexcept;
    const_iterator cbegin() const noexcept;
    const_iterator cend() const noexcept;
    const_reverse_iterator cbegin() const noexcept;
    const_reverse_iterator cend() const noexcept;

  § 22.4.7.1 883
// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class InputIterator>
void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(multiset) noexcept(allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_swappable_v<Compare>);
void clear() noexcept;

template<class C2>
void merge(multiset<Key, C2, Allocator>& source);
template<class C2>
void merge(multiset<Key, C2, Allocator>&& source);
template<class C2>
void merge(set<Key, C2, Allocator>& source);
template<class C2>
void merge(set<Key, C2, Allocator>&& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// set operations
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
template<class K> iterator find(const K& x);
template<class K> const_iterator find(const K& x) const;

size_type count(const key_type& x) const;
template<class K> size_type count(const K& x) const;

bool contains(const key_type& x) const;
template<class K> bool contains(const K& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
template<class K> iterator lower_bound(const K& x);
template<class K> const_iterator lower_bound(const K& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
template<class K> iterator upper_bound(const K& x);
template<class K> const_iterator upper_bound(const K& x) const;
pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
template<class K>
  pair<iterator, iterator> equal_range(const K& x);
template<class K>
  pair<const_iterator, const_iterator> equal_range(const K& x) const;
};
template<class InputIterator,
  class Compare = less<\text{iter-value-type}<\text{InputIterator>>},
  class Allocator = allocator<\text{iter-value-type}<\text{InputIterator>>>
multiset(InputIterator, InputIterator,
  Compare = Compare(), Allocator = Allocator())
  -> multiset<\text{iter-value-type}<\text{InputIterator}>, Compare, Allocator>);
template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
multiset(initializer_list<Key>, Compare = Compare(), Allocator = Allocator())
  -> multiset<Key, Compare, Allocator>;
template<class InputIterator, class Allocator>
multiset(InputIterator, InputIterator, Allocator)
  -> multiset<\text{iter-value-type}<\text{InputIterator}>,
    less<\text{iter-value-type}<\text{InputIterator}>, Allocator>;
template<class Key, class Allocator>
multiset(initializer_list<Key>, Allocator) -> multiset<Key, less<Key>, Allocator>;

// swap
void swap(multiset<Key, Compare, Allocator>& x,
  multiset<Key, Compare, Allocator>& y)
  noexcept(noexcept(x.swap(y)));

\section*{22.4.7.2 Constructors} \hfill\[multiset.cons\]

\subsection*{explicit multiset(const Compare& comp, const Allocator& = Allocator());}
\textbf{Effects}: Constructs an empty \texttt{multiset} using the specified comparison object and allocator.
\textbf{Complexity}: Constant.

\subsection*{template<class InputIterator>
  multiset(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& = Allocator());}
\textbf{Effects}: Constructs an empty \texttt{multiset} using the specified comparison object and allocator, and inserts elements from the range \texttt{[first, last)}.
\textbf{Complexity}: Linear in \(N\) if the range \texttt{[first, last)} is already sorted using \texttt{comp} and otherwise \(N \log N\), where \(N\) is \texttt{last} - \texttt{first}.

\section*{22.4.7.3 Erasure} \hfill\[multiset.erosure\]

\subsection*{template<class Key, class Compare, class Allocator, class Predicate>
  typename multiset<Key, Compare, Allocator>::size_type
  erase_if(multiset<Key, Compare, Allocator>& c, Predicate pred);}
\textbf{Effects}: Equivalent to:
\begin{verbatim}
  auto original_size = c.size();
  for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
      i = c.erase(i);
    } else {
      ++i;
    }
  }
\end{verbatim}
22.5 Unordered associative containers

22.5.1 In general

1 The header `<unordered_map>` defines the class templates `unordered_map` and `unordered_multimap`; the header `<unordered_set>` defines the class templates `unordered_set` and `unordered_multiset`.

2 The exposition-only alias templates `iter-value-type`, `iter-key-type`, `iter-mapped-type`, and `iter-to-alloc-type` defined in 22.4.1 may appear in deduction guides for unordered containers.

22.5.2 Header `<unordered_map>` synopsis

```cpp
#include <compare>     // see 17.11.1
#include <initializer_list>  // see 17.10.2

namespace std {
    // 22.5.4, class template unordered_map
    template<class Key,
             class T,
             class Hash = hash<Key>,
             class Pred = equal_to<Key>,
             class Alloc = allocator<pair<const Key, T>>>
    class unordered_map;

    // 22.5.5, class template unordered_multimap
    template<class Key,
             class T,
             class Hash = hash<Key>,
             class Pred = equal_to<Key>,
             class Alloc = allocator<pair<const Key, T>>>
    class unordered_multimap;

    template<class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);

    template<class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);

    template<class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
              unordered_map<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

    template<class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

    template<class K, class T, class H, class P, class A, class Predicate>
    typename unordered_map<K, T, H, P, A>::size_type
    erase_if(unordered_map<K, T, H, P, A>& c, Predicate pred);

    template<class K, class T, class H, class P, class A, class Predicate>
    typename unordered_multimap<K, T, H, P, A>::size_type
    erase_if(unordered_multimap<K, T, H, P, A>& c, Predicate pred);

    namespace pmr {
        template<class Key,
                 class T,
                 class Hash = hash<Key>,
                 class Pred = equal_to<Key>>
        using unordered_map =
```
std::unordered_map<Key, T, Hash, Pred,
    polymorphic_allocator<pair<const Key, T>>>;

template<class Key,
    class T,
    class Hash = hash<Key>,
    class Pred = equal_to<Key>>
using unordered_multimap =
    std::unordered_multimap<Key, T, Hash, Pred,
    polymorphic_allocator<pair<const Key, T>>>;

22.5.3 Header <unordered_set> synopsis

#include <compare>  // see 17.11.1
#include <initializer_list>  // see 17.10.2

namespace std {
    // 22.5.6, class template unordered_set
    template<class Key,
        class Hash = hash<Key>,
        class Pred = equal_to<Key>,
        class Alloc = allocator<Key>>
    class unordered_set;

    // 22.5.7, class template unordered_multiset
    template<class Key,
        class Hash = hash<Key>,
        class Pred = equal_to<Key>,
        class Alloc = allocator<Key>>
    class unordered_multiset;

    template<class Key, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_set<Key, Hash, Pred, Alloc>& a,
        const unordered_set<Key, Hash, Pred, Alloc>& b);

    template<class Key, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_multiset<Key, Hash, Pred, Alloc>& a,
        const unordered_multiset<Key, Hash, Pred, Alloc>& b);

    template<class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
        unordered_set<Key, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

    template<class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
        unordered_multiset<Key, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

    template<class K, class H, class P, class A, class Predicate>
    typename unordered_set<K, H, P, A>::size_type
    erase_if(unordered_set<K, H, P, A>& c, Predicate pred);

    template<class K, class H, class P, class A, class Predicate>
    typename unordered_multiset<K, H, P, A>::size_type
    erase_if(unordered_multiset<K, H, P, A>& c, Predicate pred);

    namespace pmr {
        template<class Key,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>>
        using unordered_set = std::unordered_set<Key, Hash, Pred,
            polymorphic_allocator<Key>>;

§ 22.5.3
template<class Key, 
    class Hash = hash<Key>, 
    class Pred = equal_to<Key>>
using unordered_multiset = std::unordered_multiset<Key, Hash, Pred, 
polymorphic_allocator<Key>>;

22.5.4 Class template unordered_map

22.5.4.1 Overview

1 An unordered_map is an unordered associative container that supports unique keys (an unordered_map contains at most one of each key value) and that associates values of another type mapped_type with the keys. The unordered_map class supports forward iterators.

2 An unordered_map meets all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 76). It provides the operations described in the preceding requirements table for unique keys; that is, an unordered_map supports the a_uniq operations in that table, not the a_eq operations. For an unordered_map<Key, T> the key type is Key, the mapped type is T, and the value type is pair<const Key, T>.

3 Subclause 22.5.4 only describes operations on unordered_map that are not described in one of the requirement tables, or for which there is additional semantic information.

namespace std {
    template<class Key, 
        class T, 
        class Hash = hash<Key>, 
        class Pred = equal_to<Key>, 
        class Allocator = allocator<pair<const Key, T>>>
    class unordered_map {
public:
    // types
    using key_type = Key;
    using mapped_type = T;
    using value_type = pair<const Key, T>;
    using hasher = Hash;
    using key_equal = Pred;
    using allocator_type = Allocator;
    using pointer = typename allocator_traits<Allocator>::pointer;
    using const_pointer = typename allocator_traits<Allocator>::const_pointer;
    using reference = value_type&;
    using const_reference = const value_type&;
    using size_type = implementation-defined; // see 22.2
    using difference_type = implementation-defined; // see 22.2

    using iterator = implementation-defined; // see 22.2
    using const_iterator = implementation-defined; // see 22.2
    using local_iterator = implementation-defined; // see 22.2
    using const_local_iterator = implementation-defined; // see 22.2
    using node_type = unspecified;
    using insert_return_type = insert_return_type<iterator, node_type>;

    // 22.5.4.2, construct/copy/destroy
    unordered_map();
    explicit unordered_map(size_type n, 
        const hasher& hf = hasher(), 
        const key_equal& eql = key_equal(), 
        const allocator_type& a = allocator_type());

    template<class InputIterator>
    unordered_map(InputIterator f, InputIterator l, 
        size_type n = see below, 
        const hasher& hf = hasher(), 
        const key_equal& eql = key_equal(), 
        const allocator_type& a = allocator_type());

§ 22.5.4.1
unordered_map(const unordered_map&);
unordered_map(unordered_map&&);
explicit unordered_map(const Allocator&);
unordered_map(const unordered_map&, const Allocator&);
unordered_map(initializer_list<value_type> il,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
unordered_map(size_type n, const Allocator& a)
    : unordered_map(n, hasher(), key_equal(), a) { }
unordered_map(size_type n, const hasher& hf, const allocator_type& a)
    : unordered_map(n, hf, key_equal(), a) { }

 template<class InputIterator>
 unordered_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
    : unordered_map(f, l, n, hasher(), key_equal(), a) { }

 template<class InputIterator>
 unordered_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,
    const allocator_type& a)
    : unordered_map(f, l, n, hf, key_equal(), a) { }
unordered_map(initializer_list<value_type> il, size_type n, const allocator_type& a)
    : unordered_map(il, n, hasher(), key_equal(), a) { }
unordered_map(initializer_list<value_type> il, size_type n, const hasher& hf,
    const allocator_type& a)
    : unordered_map(il, n, hf, key_equal(), a) { }
unordered_map();
unordered_map& operator=(const unordered_map&);
unordered_map& operator=(unordered_map&&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
    is_nothrow_move_assignable_v<Hash> &&
    is_nothrow_move_assignable_v<Pred>);
unordered_map& operator=(initializer_list<value_type>);

 allocator_type get_allocator() const noexcept;

 // iterators
 iterator begin() noexcept;
 const_iterator begin() const noexcept;
 iterator end() noexcept;
 const_iterator end() const noexcept;
 const_iterator cbegin() const noexcept;
 const_iterator cend() const noexcept;

 // capacity
 [[nodiscard]] bool empty() const noexcept;
 size_type size() const noexcept;
 size_type max_size() const noexcept;

 // 22.5.4.4, modifiers
 template<class... Args> pair<iterator, bool> emplace(Args&&... args);
 template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
 pair<iterator, bool> insert(const value_type& obj);
 pair<iterator, bool> insert(value_type&& obj);
 template<class P> pair<iterator, bool> insert(P&& obj);
 iterator insert(const_iterator hint, const value_type& obj);
 iterator insert(const_iterator hint, value_type&& obj);
 template<class P> iterator insert(const_iterator hint, P&& obj);
 template<class InputIterator> void insert(InputIterator first, InputIterator last);
 void insert(initializer_list<value_type>);

 node_type extract(const_iterator position);
 node_type extract(const key_type& x);
 insert_return_type insert(node_type&& nh);
 iterator insert(const_iterator hint, node_type&& nh);
template<class... Args>
  pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
  pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void swap(unordered_map&) noexcept(allocator_traits<Allocator>::is_always_equal::value &&
  is_nothrow_swappable_v<Hash> &&
  is_nothrow_swappable_v<Pred>);
void clear() noexcept;

template<class H2, class P2>
  void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);

// observers
  hasher hash_function() const;
  key_equal key_eq() const;

// map operations
  iterator find(const key_type& k);
  const_iterator find(const key_type& k) const;
  template<class K>
    iterator find(const K& k);
  template<class K>
    const_iterator find(const K& k) const;
  template<class K>
    size_type count(const key_type& k) const;
  template<class K>
    size_type count(const K& k) const;
  bool contains(const key_type& k) const;
  template<class K>
    bool contains(const K& k) const;
  pair<iterator, iterator> equal_range(const key_type& k);
  pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
  template<class K>
    pair<iterator, iterator> equal_range(const K& k);
  template<class K>
    pair<const_iterator, const_iterator> equal_range(const K& k) const;

// 22.5.4.3, element access
  mapped_type& operator[](const key_type& k);
  mapped_type& operator[](key_type&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;

bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);

};

template<class InputIterator,
    class Hash = hash<iter-key-type<InputIterator>>,
    class Pred = equal_to<iter-key-type<InputIterator>>,
    class Allocator = allocator<iter-to-alloc-type<InputIterator>>>
unordered_map<InputIterator, InputIterator, typename see below::size_type = see below,
    Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash, Pred, Allocator>;

template<class Key, class T, class Hash = hash<Key>,
    class Pred = equal_to<Key>, class Allocator = allocator<pair<const Key, T>>>
unordered_map(initializer_list<pair<Key, T>>, typename see below::size_type = see below,
    Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_map<Key, T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_map<InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
    hash<iter-key-type<InputIterator>>,
    equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_map<InputIterator, InputIterator, Allocator)
-> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
    hash<iter-key-type<InputIterator>>,
    equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_map<InputIterator, InputIterator, typename see below::size_type, Hash, Allocator)
-> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
    equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
unordered_map(initializer_list<pair<Key, T>>, typename see below::size_type, Allocator)
-> unordered_map<Key, T, hash<Key>, equal_to<Key>, Allocator>;

template<class Key, class T, class Allocator>
unordered_map(initializer_list<pair<Key, T>>, Allocator)
-> unordered_map<Key, T, hash<Key>, equal_to<Key>, Allocator>;
template<class Key, class T, class Hash, class Allocator>
unordered_map(initializer_list<pair<Key, T>>, typename
::size_type, Hash,
Allocator) -> unordered_map<Key, T, Hash, equal_to<Key>, Allocator>;

// swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
unordered_map<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));
}

A size_type parameter type in an unordered_map deduction guide refers to the size_type member type of the type deduced by the deduction guide.

22.5.4.2 Constructors

unordered_map() : unordered_map(size_type(see below)) { }
explicit unordered_map(size_type n,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_map using the specified hash function, key equality predicate, and allocator, and using at least n buckets. For the default constructor, the number of buckets is implementation-defined. max_load_factor() returns 1.0.

Complexity: Constant.

template<class InputIterator>
unordered_map(InputIterator f, InputIterator l,
size_type n = see below,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());
unordered_map(initializer_list<value_type> il,
size_type n = see below,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_map using the specified hash function, key equality predicate, and allocator, and using at least n buckets. If n is not provided, the number of buckets is implementation-defined. Then inserts elements from the range [f, l) for the first form, or from the range [il.begin(), il.end()) for the second form. max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.

22.5.4.3 Element access

mapped_type& operator[](const key_type& k);

Effects: Equivalent to: return try_emplace(k).first->second;

mapped_type& operator[](key_type&& k);

Effects: Equivalent to: return try_emplace(move(k)).first->second;

mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;

Returns: A reference to x.second, where x is the (unique) element whose key is equivalent to k.

Throws: An exception object of type out_of_range if no such element is present.

22.5.4.4 Modifiers

template<class P>
pair<iterator, bool> insert(P&& obj);

Constraints: is_constructible_v=value_type, P&&> is true.
Effects: Equivalent to: return emplace(std::forward<P>(obj));

```cpp
template<class P>
iterator insert(const_iterator hint, P&& obj);
```

Constraints: is_constructible_v<value_type, P&&> is true.

Effects: Equivalent to: return emplace_hint(hint, std::forward<P>(obj));

```cpp
template<class... Args>
pair<iterator, bool> try_emplace(const_iterator hint, const key_type& k, Args&&... args);
```

Preconditions: value_type is Cpp17EmplaceConstructible into unordered_map from piecewise_construct, forward_as_tuple(k), forward_as_tuple(std::forward<Args>(args)...)

Effects: If the map already contains an element whose key is equivalent to k, there is no effect. Otherwise inserts an object of type value_type constructed with piecewise_construct, forward_as_tuple(k), forward_as_tuple(std::forward<Args>(args)...).

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

```cpp
template<class... Args>
pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
```

Preconditions: value_type is Cpp17EmplaceConstructible into unordered_map from piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...)

Effects: If the map already contains an element whose key is equivalent to k, there is no effect. Otherwise inserts an object of type value_type constructed with piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...).

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

```cpp
template<class M>
pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
```

Mandates: is_assignable_v<mapped_type&, M&&> is true.

Preconditions: value_type is Cpp17EmplaceConstructible into unordered_map from k, std::forward<M>(obj).

Effects: If the map already contains an element e whose key is equivalent to k, assigns std::forward<M>(obj) to e.second. Otherwise inserts an object of type value_type constructed with k, std::forward<M>(obj).

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

```cpp
template<class M>
pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
```

Mandates: is_assignable_v<mapped_type&, M&&> is true.

Preconditions: value_type is Cpp17EmplaceConstructible into unordered_map from std::move(k), std::forward<M>(obj).

§ 22.5.4.4
Effects: If the map already contains an element \( e \) whose key is equivalent to \( k \), assigns `std::forward<M>(obj)` to \( e.second \). Otherwise inserts an object of type `value_type` constructed with `std::move(k), std::forward<M>(obj)`.

Returns: In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to \( k \).

Complexity: The same as `emplace` and `emplace_hint`, respectively.

### 22.5.4.5 Erasure

```cpp
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_map<K, T, H, P, A>::size_type
erase_if(unordered_map<K, T, H, P, A>& c, Predicate pred);
```

Effects: Equivalent to:

```cpp
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
} return original_size - c.size();
```

### 22.5.5 Class template unordered_multimap

#### 22.5.5.1 Overview

An `unordered_multimap` is an unordered associative container that supports equivalent keys (an instance of `unordered_multimap` may contain multiple copies of each key value) and that associates values of another type `mapped_type` with the keys. The `unordered_multimap` class supports forward iterators.

An `unordered_multimap` meets all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 76). It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multimap` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multimap<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `pair<const Key, T>`.

Subclause 22.5.5 only describes operations on `unordered_multimap` that are not described in one of the requirement tables, or for which there is additional semantic information.

```cpp
namespace std {
    template<class Key, class T, class Hash = hash<Key>,
             class Pred = equal_to<Key>,
             class Allocator = allocator<pair<const Key, T>>>
    class unordered_multimap {
        public:
            // types
            using key_type = Key;
            using mapped_type = T;
            using value_type = pair<const Key, T>;
            using hasher = Hash;
            using key_equal = Pred;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using difference_type = implementation-defined; // see 22.2
            using size_type = implementation-defined; // see 22.2
            using iterator = implementation-defined; // see 22.2
            using const_iterator = implementation-defined; // see 22.2
    }
}
```
using local_iterator = implementation-defined; // see 22.2
using const_local_iterator = implementation-defined; // see 22.2
using node_type = unspecified;

// 22.5.5.2, construct/copy/destroy
unordered_multimap();
explicit unordered_multimap(size_type n,  
    const hasher& hf = hasher(),  
    const key_equal& eql = key_equal(),  
    const allocator_type& a = allocator_type());

template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l,  
    size_type n = see below,  
    const hasher& hf = hasher(),  
    const key_equal& eql = key_equal(),  
    const allocator_type& a = allocator_type());
unordered_multimap(const unordered_multimap&);
unordered_multimap(unordered_multimap&&);
explicit unordered_multimap(const Allocator&);
unordered_multimap(const unordered_multimap&, const Allocator&);
unordered_multimap(unordered_multimap&, const Allocator&);
unordered_multimap(initializer_list<value_type> il,  
    size_type n = see below,  
    const hasher& hf = hasher(),  
    const key_equal& eql = key_equal(),  
    const allocator_type& a = allocator_type());
unordered_multimap(size_type n, const allocator_type& a)  
    : unordered_multimap(n, hasher(), key_equal(), a) { }  
unordered_multimap(size_type n, const hasher& hf, const allocator_type& a)  
    : unordered_multimap(n, hf, key_equal(), a) { }  
template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l, size_type n, const allocator_type& a)  
    : unordered_multimap(f, l, n, hasher(), key_equal(), a) { }  
template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l, size_type n, const hasher& hf,  
    const allocator_type& a)  
    : unordered_multimap(f, l, n, hf, key_equal(), a) { }  
unordered_multimap(initializer_list<value_type> il, size_type n, const allocator_type& a)  
    : unordered_multimap(il, n, hasher(), key_equal(), a) { }  
unordered_multimap(initializer_list<value_type> il, size_type n, const hasher& hf,  
    const allocator_type& a)  
    : unordered_multimap(il, n, hf, key_equal(), a) { }  
~unordered_multimap();
unordered_multimap operator=(const unordered_multimap&);
unordered_multimap operator=(unordered_multimap&&);
nothrow(allocator_traits<Allocator>::is_always_equal::value &&  
    is_nothrow_move_assignable_v<Hash> &&  
    is_nothrow_move_assignable_v<Pred>);  
unordered_multimap operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

§ 22.5.5.1
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class P> iterator insert(const_iterator hint, P&& obj);
template<class... Args> iterator insert_hint(const_iterator position, Args&&... args);
iterator insert(const_iterator hint, Args&&... args);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);
iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void swap(unordered_multimap&);  
    noexcept(allocator_traits<Allocator>::is_always_equal::value &
        is_nothrow_swappable_v<Hash> &&
        is_nothrow_swappable_v<Pred>);
void clear() noexcept;

template<class H2, class P2>
void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);

hasher hash_function() const;
key_equal key_eq() const;

// map operations
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
    iterator find(const K& k);
template<class K>
    const_iterator find(const K& k) const;
size_type count(const key_type& k) const;
template<class K>
size_type count(const K& k) const;
bool contains(const key_type& k) const;
template<class K>
bool contains(const K& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
pair<iterator, iterator> equal_range(const K& k);
template<class K>
pair<const_iterator, const_iterator> equal_range(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

template<class InputIterator,
class Hash = hash<iter-key-type<InputIterator>>,
class Pred = equal_to<iter-key-type<InputIterator>>,
class Allocator = allocator<iter-to-alloc-type<InputIterator>>> unordered_multimap(InputIterator, InputIterator,
typename see below::size_type = see below,
Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
Hash, Pred, Allocator>;

template<class Key, class T, class Hash = hash<Key>,
class Pred = equal_to<Key>, class Allocator = allocator<pair<const Key, T>>> unordered_multimap(initializer_list<pair<Key, T>>,
type_name see below::size_type = see below,
Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_multimap<Key, T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_multimap<InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
hash<iter-key-type<InputIterator>>,
equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_multimap<InputIterator, InputIterator, Allocator)
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
hash<iter-key-type<InputIterator>>,
equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_multimap<InputIterator, InputIterator, typename see below::size_type, Hash,
Allocator)
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
unordered_multimap(initializer_list<pair<Key, T>>, typename see below::size_type,
Allocator)
-> unordered_multimap<Key, T, hash<Key>, equal_to<Key>, Allocator>;

template<class Key, class T, class Allocator>
unordered_multimap(initializer_list<pair<Key, T>>, Allocator)
-> unordered_multimap<Key, T, hash<Key>, equal_to<Key>, Allocator>;

template<class Key, class Hash, class Allocator>
unordered_multimap(initializer_list<pair<Key, T>>, typename see below::size_type,
Hash, Allocator)
-> unordered_multimap<Key, T, Hash, equal_to<Key>, Allocator>;
void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x, unordered_multimap<Key, T, Hash, Pred, Alloc>& y) noexcept(noexcept(x.swap(y)));}

A size_type parameter type in an unordered_multimap deduction guide refers to the size_type member type of the type deduced by the deduction guide.

22.5.5.2 Constructors

unordered_multimap() : unordered_multimap(size_type(see below)) { }
explicit unordered_multimap(size_type n, const hasher& hf = hasher(), const key_equal& eql = key_equal(), const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multimap using the specified hash function, key equality predicate, and allocator, and using at least n buckets. For the default constructor, the number of buckets is implementation-defined. max_load_factor() returns 1.0.

Complexity: Constant.

template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l, size_type n = see below, const hasher& hf = hasher(), const key_equal& eql = key_equal(), const allocator_type& a = allocator_type());
unordered_multimap(initializer_list<value_type> il, size_type n = see below, const hasher& hf = hasher(), const key_equal& eql = key_equal(), const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multimap using the specified hash function, key equality predicate, and allocator, and using at least n buckets. If n is not provided, the number of buckets is implementation-defined. Then inserts elements from the range [f, l) for the first form, or from the range [il.begin(), il.end()) for the second form. max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.

22.5.5.3 Modifiers

template<class P>
iterator insert(P&& obj);

Constraints: is_constructible_v<value_type, P&&> is true.

Effects: Equivalent to: return emplace(std::forward<P>(obj));

template<class P>
iterator insert(const_iterator hint, P&& obj);

Constraints: is_constructible_v<value_type, P&&> is true.

Effects: Equivalent to: return emplace_hint(hint, std::forward<P>(obj));

22.5.5.4 Erasure

template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_multimap<K, T, H, P, A>::size_type
erase_if(unordered_multimap<K, T, H, P, A>& c, Predicate pred);

Effects: Equivalent to:

auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}

§ 22.5.4
22.5.6 Class template unordered_set

22.5.6.1 Overview

An unordered_set is an unordered associative container that supports unique keys (an unordered_set contains at most one of each key value) and in which the elements’ keys are the elements themselves. The unordered_set class supports forward iterators.

An unordered_set meets all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 76). It provides the operations described in the preceding requirements table for unique keys; that is, an unordered_set supports the a_uniq operations in that table, not the a_eq operations. For an unordered_set<Key> the key type and the value type are both Key. The iterator and const_iterator types are both constant iterator types. It is unspecified whether they are the same type.

Subclause 22.5.6 only describes operations on unordered_set that are not described in one of the requirement tables, or for which there is additional semantic information.

```cpp
namespace std {
    template<class Key, class Hash = hash<Key>, class Pred = equal_to<Key>, class Allocator = allocator<Key>>
    class unordered_set {
        public:
            // types
            using key_type = Key;
            using value_type = Key;
            using hasher = Hash;
            using key_equal = Pred;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation-defined; // see 22.2
            using difference_type = implementation-defined; // see 22.2
            using iterator = implementation-defined; // see 22.2
            using const_iterator = implementation-defined; // see 22.2
            using local_iterator = implementation-defined; // see 22.2
            using const_local_iterator = implementation-defined; // see 22.2
            using node_type = unspecified;
            using insert_return_type = insert-return-type<iterator, node_type>;

            // 22.5.6.2, construct/copy/destroy
            unordered_set();
            explicit unordered_set(size_type n,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
            template<class InputIterator>
            unordered_set(InputIterator f, InputIterator l,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
            unordered_set(const unordered_set&);
            unordered_set(unordered_set&&);
            explicit unordered_set(const Allocator&);
            unordered_set(const unordered_set&, const Allocator&);

    }
```
unordered_set(unordered_set&&, const Allocator&);
unordered_set(initializer_list<value_type> il,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
unordered_set(size_type n, const allocator_type& a)
  : unordered_set(n, hasher(), key_equal(), a) {}  
unordered_set(size_type n, const hasher& hf, const allocator_type& a)
  : unordered_set(n, hf, key_equal(), a) {}  

template<class InputIterator>
unordered_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
  : unordered_set(f, l, n, hasher(), key_equal(), a) {}  

unordered_set(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_set(il, size_type n, hasher(), key_equal(), a) {}  
unordered_set(initializer_list<value_type> il, size_type n, const hasher& hf,
    const allocator_type& a)
  : unordered_set(il, n, hf, key_equal(), a) {}  

unordered_set<InputIterator f, InputIterator l, size_type n, const allocator_type& a)
  : unordered_set(f, l, n, hf, key_equal(), a) {}  

unordered_set& operator=(const unordered_set&);
unordered_set& operator=(unordered_set&&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
            is_nothrow_move_assignable_y<Hash> &&
            is_nothrow_move_assignable_y<Pred>);
unordered_set& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<
class... Args>
pair<
iterator,
bool>
emplace(Args&&... args);
template<
class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
pair<
iterator,
bool>
insert(const value_type& obj);
pair<
iterator,
bool>
insert(value_type&& obj);

iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);

void insert(InputIterator first, InputIterator last);

node_type extract(const_iterator position);
ode_type extract(const key_type& x);
inser et_return_type insert(node_type& nh);
iterator insert(const_iterator hint, node_type& nh);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void swap(unordered_set&) noexcept(allocator_traits<Allocator>::is_always_equal::value &&
   is_nothrow_swappable_v<Hash> &&
   is_nothrow_swappable_v<Pred>);

void clear() noexcept;

template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// set operations
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator find(const K& k);
template<class K>
const_iterator find(const K& k) const;
size_type count(const key_type& k) const;
size_type count(const K& k) const;
bool contains(const key_type& k) const;
bool contains(const K& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
pair<iterator, iterator> equal_range(const K& k);
template<class K>
pair<const_iterator, const_iterator> equal_range(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket(size_type n) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

template<class InputIterator,
   class Hash = hash<typename iter_value_type<InputIterator>>::size_type = see below,
   class Pred = equal_to<typename iter_value_type<InputIterator>>::size_type = see below,
   class Allocator = allocator<typename iter_value_type<InputIterator>>::size_type = see below,
   class InputIterator,
   class Hash = hash<typename iter_value_type<InputIterator>>::size_type = see below,
   class Pred = equal_to<typename iter_value_type<InputIterator>>::size_type = see below,
   class Allocator = allocator<typename iter_value_type<InputIterator>>::size_type = see below,
   class InputIterator,
   class Hash = hash<typename iter_value_type<InputIterator>>::size_type = see below,
   class Pred = equal_to<typename iter_value_type<InputIterator>>::size_type = see below,
   class Allocator = allocator<typename iter_value_type<InputIterator>>::size_type = see below,
   class InputIterator,
   class Hash = hash<typename iter_value_type<InputIterator>>::size_type = see below,
   class Pred = equal_to<typename iter_value_type<InputIterator>>::size_type = see below,
   class Allocator = allocator<typename iter_value_type<InputIterator>>::size_type = see below,
   class InputIterator,
   class Hash = hash<typename iter_value_type<InputIterator>>::size_type = see below,
Hash = Hash(), Pred = Pred(), Allocator = Allocator();
-> unordered_set<iter-value-type<InputIterator>,
   Hash, Pred, Allocator>;

template<class T, class Hash = hash<T>,
   class Pred = equal_to<T>, class Allocator = allocator<T>>
unordered_set(initializer_list<T>, typename see below::size_type = see below,
   Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_set<T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_set(InputIterator, InputIterator, typename see below::size_type,
   Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_set<T, Hash, Pred, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_set(InputIterator, InputIterator, typename see below::size_type,
   Hash, allocator_type) 
-> unordered_set<T, Hash, equal_to<T>, Allocator>;

template<class T, class Allocator>
unordered_set(initializer_list<T>, typename see below::size_type, Allocator)
-> unordered_set<T, hash<T>, equal_to<T>, Allocator>;

// swap

template<class Key, class Hash, class Pred, class Alloc>
void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
   unordered_set<Key, Hash, Pred, Alloc>& y)
   noexcept(noexcept(x.swap(y)));
}

A size_type parameter type in an unordered_set deduction guide refers to the size_type member type of the type deduced by the deduction guide.

22.5.6.2 Constructors

unordered_set() : unordered_set(size_type(see below)) {} 
explicit unordered_set(size_type n,
   const hasher& hf = hasher(),
   const key_equal& eql = key_equal(),
   const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_set using the specified hash function, key equality predicate, and allocator, and using at least n buckets. For the default constructor, the number of buckets is implementation-defined. max_load_factor() returns 1.0.

Complexity: Constant.

template<class InputIterator>
unordered_set(InputIterator f, InputIterator l,
   size_type n = see below,
   const hasher& hf = hasher(),
   const key_equal& eql = key_equal(),
   const allocator_type& a = allocator_type());
const allocator_type& a = allocator_type();

Effects: Constructs an empty unordered_set using the specified hash function, key equality predicate, and allocator, and using at least n buckets. If n is not provided, the number of buckets is implementation-defined. Then inserts elements from the range [f, l) for the first form, or from the range [il.begin(), il.end()) for the second form. max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.

22.5.6.3 Erasure

template<class K, class H, class P, class A, class Predicate>
typename unordered_set<K, H, P, A>::size_type
erase_if(unordered_set<K, H, P, A>& c, Predicate pred);

Effects: Equivalent to:

auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
return original_size - c.size();

22.5.7 Class template unordered_multiset

22.5.7.1 Overview

An unordered_multiset is an unordered associative container that supports equivalent keys (an instance of unordered_multiset may contain multiple copies of the same key value) and in which each element’s key is the element itself. The unordered_multiset class supports forward iterators.

An unordered_multiset meets all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 76). It provides the operations described in the preceding requirements table for equivalent keys; that is, an unordered_multiset supports the a_eq operations in that table, not the a_uniq operations. For an unordered_multiset<Key> the key type and the value type are both Key. The iterator and const_iterator types are both constant iterator types. It is unspecified whether they are the same type.

Subclause 22.5.7 only describes operations on unordered_multiset that are not described in one of the requirement tables, or for which there is additional semantic information.

namespace std {
    template<class Key,
        class Hash = hash<Key>,
        class Pred = equal_to<Key>,
        class Allocator = allocator<Key>>
    class unordered_multiset {
        public:
            // types
            using key_type = Key;
            using value_type = Key;
            using hasher = Hash;
            using key_equal = Pred;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation_defined; // see 22.2
            using difference_type = implementation_defined; // see 22.2

            using iterator = implementation_defined; // see 22.2
            using const_iterator = implementation_defined; // see 22.2
            using local_iterator = implementation_defined; // see 22.2

§ 22.5.7.1 903
using const_local_iterator = implementation-defined; // see 22.2
using node_type = unspecified;

// 22.5.7.2, construct/copy/destroy
unordered_multiset();
explicit unordered_multiset(size_type n,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
template<class InputIterator>
unordered_multiset(InputIterator f, InputIterator l,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
unordered_multiset(const unordered_multiset&);
unordered_multiset(unordered_multiset&&);
explicit unordered_multiset(const Allocator&);
unordered_multiset(const unordered_multiset&, const Allocator&);
unordered_multiset(unordered_multiset&&, const Allocator&);
unordered_multiset(initializer_list<value_type> il,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
unordered_multiset(size_type n, const hasher& hf, const allocator_type& a)
    : unordered_multiset(n, hf, key_equal(), a) { }
unordered_multiset(size_type n, const hasher& hf, const node_type& a)
    : unordered_multiset(n, hf, key_equal(), a) { }
template<class InputIterator>
unordered_multiset(InputIterator f, InputIterator l, size_type n,
    const allocator_type& a) :
    unordered_multiset(f, l, n, hasher(), key_equal(), a) { }
unordered_multiset(initializer_list<value_type> il, size_type n,
    const allocator_type& a)
    : unordered_multiset(il, n, hasher(), key_equal(), a) { }
-unordered_multiset();
unordered_multiset& operator=(const unordered_multiset&);
unordered_multiset& operator=(unordered_multiset&&);
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
        is_nothrow_moveAssignable_v<Hash> &&
        is_nothrow_moveAssignable_v<Pred>);
unordered_multiset& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterates
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

§ 22.5.7.1
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);
iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void swap(unordered_multiset&);

noexcept((allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_swappable_v<Hash> &&
           is_nothrow_swappable_v<Pred>));
void clear() noexcept;

template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// set operations
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator find(const K& k);
template<class K>
const_iterator find(const K& k) const;
size_type count(const key_type& k) const;
template<class K>
size_type count(const K& k) const;
bool contains(const key_type& k) const;
template<class K>
bool contains(const K& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;

local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

template<class InputIterator,
class Hash = hash<iter-value-type<InputIterator>>,
class Pred = equal_to<iter-value-type<InputIterator>>,
class Allocator = allocator<iter-value-type<InputIterator>>>
unordered_multiset(InputIterator, InputIterator, see below::{size_type = see below,
Hash = Hash(), Pred = Pred(), Allocator = Allocator()})
-> unordered_multiset<iter-value-type<InputIterator>>,
Hash, Pred, Allocator>;

template<class T, class Hash = hash<T>,
class Pred = equal_to<T>, class Allocator = allocator<T>>
unordered_multiset(initializer_list<T>, typename see below::{size_type = see below,
Hash = Hash(), Pred = Pred(), Allocator = Allocator()})
-> unordered_multiset<T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_multiset(InputIterator, InputIterator, typename see below::{size_type, Allocator})
-> unordered_multiset<iter-value-type<InputIterator>>,
hash<iter-value-type<InputIterator>>,
equal_to<iter-value-type<InputIterator>>,
Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_multiset(InputIterator, InputIterator, typename see below::{size_type,
Hash, Allocator})
-> unordered_multiset<iter-value-type<InputIterator>>, Hash,
equal_to<iter-value-type<InputIterator>>,
Allocator>;

template<class T, class Allocator>
unordered_multiset(initializer_list<T>, typename see below::{size_type, Allocator})
-> unordered_multiset<T, hash<T>, equal_to<T>, Allocator>;

template<class T, class Hash, class Allocator>
unordered_multiset(initializer_list<T>, typename see below::{size_type, Hash, Allocator})
-> unordered_multiset<T, Hash, equal_to<T>, Allocator>;

// swap
template<class Key, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
unordered_multiset<Key, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));
}

A size_type parameter type in an unordered_multiset deduction guide refers to the size_type member

type of the type deduced by the deduction guide.

22.5.7.2 Constructors [unord.multiset.cnstr]

unordered_multiset() : unordered_multiset(size_type(see below)) { }
explicit unordered_multiset(size_type n,  
    const hasher& hf = hasher(),  
    const key_equal& eql = key_equal(),  
    const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multiset using the specified hash function, key equality predicate, and allocator, and using at least n buckets. For the default constructor, the number of buckets is implementation-defined. max_load_factor() returns 1.0.

Complexity: Constant.

template<class InputIterator>  
unordered_multiset(InputIterator f, InputIterator l,  
    size_type n = see below,  
    const hasher& hf = hasher(),  
    const key_equal& eql = key_equal(),  
    const allocator_type& a = allocator_type());

unordered_multiset(initializer_list<value_type> il,  
    size_type n = see below,  
    const hasher& hf = hasher(),  
    const key_equal& eql = key_equal(),  
    const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multiset using the specified hash function, key equality predicate, and allocator, and using at least n buckets. If n is not provided, the number of buckets is implementation-defined. Then inserts elements from the range [f, l) for the first form, or from the range [il.begin(), il.end()) for the second form. max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.

22.5.7.3 Erasure

template<class K, class H, class P, class A, class Predicate>  
typename unordered_multiset<K, H, P, A>::size_type  
erase_if(unordered_multiset<K, H, P, A>& c, Predicate pred);

Effects: Equivalent to:

auto original_size = c.size();  
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);  
    } else {  
        ++i;  
    }
}  
return original_size - c.size();

22.6 Container adaptors

22.6.1 In general

The headers <queue> and <stack> define the container adaptors queue, priority_queue, and stack.

The container adaptors each take a Container template parameter, and each constructor takes a Container reference argument. This container is copied into the Container member of each adaptor. If the container takes an allocator, then a compatible allocator may be passed in to the adaptor’s constructor. Otherwise, normal copy or move construction is used for the container argument. The first template parameter T of the container adaptors shall denote the same type as Container::value_type.

For container adaptors, no swap function throws an exception unless that exception is thrown by the swap of the adaptor’s Container or Compare object (if any).

A deduction guide for a container adaptor shall not participate in overload resolution if any of the following are true:

— It has an InputIterator template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
— It has a `Compare` template parameter and a type that qualifies as an allocator is deduced for that parameter.

— It has a `Container` template parameter and a type that qualifies as an allocator is deduced for that parameter.

— It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.

— It has both `Container` and `Allocator` template parameters, and `uses_allocator_v<Container, Allocator>` is false.

### 22.6.2 Header `<queue>` synopsis

```cpp
#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.2

namespace std {
    template<class T, class Container = deque<T>> class queue;
    template<class T, class Container>
    bool operator===(const queue<T, Container>& x, const queue<T, Container>& y);
    template<class T, class Container>
    bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);
    template<class T, class Container>
    bool operator< (const queue<T, Container>& x, const queue<T, Container>& y);
    template<class T, class Container>
    bool operator> (const queue<T, Container>& x, const queue<T, Container>& y);
    template<class T, class Container>
    bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);
    template<class T, class Container>
    bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);
    template<class T, three_way_comparable Container>
    compare_three_way_result_t<Container>
    operator<=>(const queue<T, Container>& x, const queue<T, Container>& y);
    template<class T, class Container>
    void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));
    template<class T, class Container, class Alloc>
    struct uses_allocator<queue<T, Container>, Alloc>;
}
```

### 22.6.3 Header `<stack>` synopsis

```cpp
#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.2

namespace std {
    template<class T, class Container = deque<T>> class stack;
    template<class T, class Container>
    bool operator===(const stack<T, Container>& x, const stack<T, Container>& y);
    template<class T, class Container>
    bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
    template<class T, class Container>
    bool operator< (const stack<T, Container>& x, const stack<T, Container>& y);
    template<class T, class Container>
    bool operator> (const stack<T, Container>& x, const stack<T, Container>& y);
    template<class T, class Container>
    bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);
    template<class T, class Container>
    bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
    template<class T, class Container, class Compare = less<typename Container::value_type>>
    class priority_queue;
    template<class T, class Container, class Compare>
    void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));
    template<class T, class Container, class Compare, class Alloc>
    struct uses_allocator<priority_queue<T, Container, Compare>, Alloc>;
}
```
template<class T, class Container>
bool operator>(const stack<T, Container>& x, const stack<T, Container>& y);

template<class T, class Container>
bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);

template<class T, class Container>
bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);

template<class T, three_way_comparable Container>
compare_three_way_result_t<Container>
operator<=>(const stack<T, Container>& x, const stack<T, Container>& y);

template<class T, class Container>
void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));

namespace std {

template<class T, class Container = deque<T>>
class queue {
public:
    using value_type = typename Container::value_type;
    using reference = typename Container::reference;
    using const_reference = typename Container::const_reference;
    using size_type = typename Container::size_type;
    using container_type = Container;

    protected:
        Container c;

    public:
        queue() : queue(Container()) {}  // 22.6.4.1.1
        explicit queue(const Container&);
        explicit queue(Container&&);
        template<class Alloc> explicit queue(const Alloc&);
        template<class Alloc> queue(const Container&, const Alloc&);
        template<class Alloc> queue(Container&&, const Alloc&);
        template<class Alloc> queue(const queue&, const Alloc&);
        template<class Alloc> queue(queue&&, const Alloc&);

        [[nodiscard]] bool empty() const { return c.empty(); }
        size_type size() const { return c.size(); }
        reference front() { return c.front(); }
        const_reference front() const { return c.front(); }
        reference back() { return c.back(); }
        const_reference back() const { return c.back(); }
        void push(const value_type& x) { c.push_back(x); }
        void push(value_type&& x) { c.push_back(std::move(x)); }
        template<class... Args>
        decltype(auto) emplace(Args&&... args)
            { return c.emplace_back(std::forward<Args>(args)...); }
        void pop() { c.pop_front(); }
        void swap(queue& q) noexcept(is_nothrow_swappable_v<Container>)
            { using std::swap; swap(c, q.c); }
    };

    template<class Container>
    queue(Container) -> queue<typename Container::value_type, Container>;
};

22.6.4 Class template queue

22.6.4.1 Definition

Any sequence container supporting operations front(), back(), push_back() and pop_front() can be used to instantiate queue. In particular, list (22.3.10) and deque (22.3.8) can be used.

namespace std {

template<class T, class Container = deque<T>>
class queue {
public:
    using value_type = typename Container::value_type;
    using reference = typename Container::reference;
    using const_reference = typename Container::const_reference;
    using size_type = typename Container::size_type;
    using container_type = Container;

    protected:
        Container c;

    public:
        queue() : queue(Container()) {}  // 22.6.4.1.1
        explicit queue(const Container&);
        explicit queue(Container&&);
        template<class Alloc> explicit queue(const Alloc&);
        template<class Alloc> queue(const Container&, const Alloc&);
        template<class Alloc> queue(Container&&, const Alloc&);
        template<class Alloc> queue(const queue&, const Alloc&);
        template<class Alloc> queue(queue&&, const Alloc&);

        [[nodiscard]] bool empty() const { return c.empty(); }
        size_type size() const { return c.size(); }
        reference front() { return c.front(); }
        const_reference front() const { return c.front(); }
        reference back() { return c.back(); }
        const_reference back() const { return c.back(); }
        void push(const value_type& x) { c.push_back(x); }
        void push(value_type&& x) { c.push_back(std::move(x)); }
        template<class... Args>
        decltype(auto) emplace(Args&&... args)
            { return c.emplace_back(std::forward<Args>(args)...); }
        void pop() { c.pop_front(); }
        void swap(queue& q) noexcept(is_nothrow_swappable_v<Container>)
            { using std::swap; swap(c, q.c); }
    };

    template<class Container>
    queue(Container) -> queue<typename Container::value_type, Container>;
};
template<class Container, class Allocator>
queue(Container, Allocator) -> queue<typename Container::value_type, Container>;

template<class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)))

template<class T, class Container, class Alloc>
struct uses_allocator<queue<T, Container>, Alloc>
: uses_allocator<Container, Alloc>::type {}

22.6.4.2 Constructors

explicit queue(const Container& cont);
1  Effects: Initializes c with cont.

explicit queue(Container&& cont);
2  Effects: Initializes c with std::move(cont).

22.6.4.3 Constructors with allocators

1 If uses_allocator_v<container_type, Alloc> is false the constructors in this subclause shall not par-
ticipate in overload resolution.

template<class Alloc> explicit queue(const Alloc& a);
2  Effects: Initializes c with a.

template<class Alloc> queue(const container_type& cont, const Alloc& a);
3  Effects: Initializes c with cont as the first argument and a as the second argument.

template<class Alloc> queue(container_type&& cont, const Alloc& a);
4  Effects: Initializes c with std::move(cont) as the first argument and a as the second argument.

template<class Alloc> queue(const queue& q, const Alloc& a);
5  Effects: Initializes c with q.c as the first argument and a as the second argument.

template<class Alloc> queue(queue&& q, const Alloc& a);
6  Effects: Initializes c with std::move(q.c) as the first argument and a as the second argument.

22.6.4.4 Operators

template<class T, class Container>
bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);
1  Returns: x.c == y.c.

template<class T, class Container>
bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);
2  Returns: x.c != y.c.

template<class T, class Container>
bool operator<(const queue<T, Container>& x, const queue<T, Container>& y);
3  Returns: x.c < y.c.

template<class T, class Container>
bool operator>(const queue<T, Container>& x, const queue<T, Container>& y);
4  Returns: x.c > y.c.

template<class T, class Container>
bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);
5  Returns: x.c <= y.c.

§ 22.6.4.4 910
template<class T, class Container>
bool operator>=(const queue<T, Container>& x,
const queue<T, Container>& y);

Returns: x.c >= y.c.

template<class T, three_way_comparable Container>
compare_three_way_result_t<Container>
operator<=>(const queue<T, Container>& x, const queue<T, Container>& y);

Returns: x.c <=> y.c.

22.6.4.5 Specialized algorithms [queue.special]

template<class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));

Constraints: is_swappable_v<Container> is true.

Effects: As if by x.swap(y).

22.6.5 Class template priority_queue [priority.queue]

22.6.5.1 Overview [priqueue.overview]

Any sequence container with random access iterator and supporting operations front(), push_back() and pop_back() can be used to instantiate priority_queue. In particular, vector (22.3.11) and deque (22.3.8) can be used. Instantiating priority_queue also involves supplying a function or function object for making priority comparisons; the library assumes that the function or function object defines a strict weak ordering (25.8).

namespace std {
    template<class T, class Container = vector<T>,
             class Compare = less<typename Container::value_type>>
class priority_queue {
public:
    using value_type = typename Container::value_type;
    using reference = typename Container::reference;
    using const_reference = typename Container::const_reference;
    using size_type = typename Container::size_type;
    using container_type = Container;
    using value_compare = Compare;

protected:
    Container c;
    Compare comp;

public:
    priority_queue() : priority_queue(Compare()) {}  
    explicit priority_queue(const Compare& x) : priority_queue(x, Container()) {}  
    priority_queue(const Compare& x, const Container&);  
    priority_queue(const Compare& x, Container&&);
    template<class InputIterator>
    priority_queue(InputIterator first, InputIterator last, const Compare& x,
                   const Container&);
    template<class InputIterator>
    priority_queue(InputIterator first, InputIterator last,
                   const Compare& x = Compare(), Container&& = Container());
    template<class Alloc> explicit priority_queue(const Alloc&);
    template<class Alloc> priority_queue(const Compare&, const Alloc&);
    template<class Alloc> priority_queue(const Compare&, const Container&,
                                         const Alloc&);
    template<class Alloc> priority_queue(priority_queue&&, const Alloc&);

    [[nodiscard]] bool empty() const { return c.empty(); }  
    size_type size() const { return c.size(); }  
    const_reference top() const { return c.front(); }  
}

§ 22.6.5.1 911
void push(const value_type& x);
void push(value_type&& x);
template<class... Args> void emplace(Args&&... args);
void pop();
void swap(priority_queue& q) noexcept(is_nothrow_swappable_v<Container> &&
  is_nothrow_swappable_v<Compare>)
  { using std::swap; swap(c, q.c); swap(comp, q.comp); }
};
template<class Compare, class Container>
priority_queue(Compare, Container) -> priority_queue<typename Container::value_type, Container, Compare>;
template<class InputIterator,
  class Compare = less<typename iterator_traits<InputIterator>::value_type>,
  class Container = vector<typename iterator_traits<InputIterator>::value_type>>
priority_queue(InputIterator, InputIterator, Compare = Compare(), Container = Container()) -> priority_queue<typename iterator_traits<InputIterator>::value_type, Container, Compare>;
template<class Compare, class Container, class Allocator>
priority_queue(Compare, Container, Allocator) -> priority_queue<typename Container::value_type, Container, Compare>;

// no equality is provided

template<class T, class Container, class Compare>
void swap(priority_queue<T, Container, Compare>& x,
  priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));

22.6.5.2 Constructors [priqueue.cons]
priority_queue(const Compare& x, const Container& y);
priority_queue(const Compare& x, Container&& y);
  
  **Preconditions:** x defines a strict weak ordering (25.8).
  
  **Effects:** Initializes comp with x and c with y (copy constructing or move constructing as appropriate); calls make_heap(c.begin(), c.end(), comp).

template<class InputIterator>
priority_queue(InputIterator first, InputIterator last, const Compare& x, const Container& y);
template<class InputIterator>
priority_queue(InputIterator first, InputIterator last, const Compare& x = Compare(),
  Container&& y = Container());
  
  **Preconditions:** x defines a strict weak ordering (25.8).
  
  **Effects:** Initializes comp with x and c with y (copy constructing or move constructing as appropriate); calls c.insert(c.end(), first, last); and finally calls make_heap(c.begin(), c.end(), comp).

22.6.5.3 Constructors with allocators [priqueue.cons.alloc]

If **uses_allocator_v<container_type, Alloc>** is **false** the constructors in this subclause shall not participate in overload resolution.

template<class Alloc> explicit priority_queue(const Alloc& a);
  
  **Effects:** Initializes c with a and value-initializes comp.

template<class Alloc> priority_queue(const Compare& compare, const Alloc& a);
  
  **Effects:** Initializes c with a and initializes comp with compare.
```cpp
template<class Alloc>
    priority_queue(const Compare& compare, const Container& cont, const Alloc& a);

Effects: Initializes c with cont as the first argument and a as the second argument, and initializes comp
with compare; calls make_heap(c.begin(), c.end(), comp).

template<class Alloc>
    priority_queue(const Compare& compare, const Container& cont, const Alloc& a);

Effects: Initializes c with std::move(cont) as the first argument and a as the second argument, and
initializes comp with compare; calls make_heap(c.begin(), c.end(), comp).

template<class Alloc> priority_queue(const priority_queue& q, const Alloc& a);

Effects: Initializes c with q.c as the first argument and a as the second argument, and initializes comp
with q.comp.

template<class Alloc> priority_queue(priority_queue&& q, const Alloc& a);

Effects: Initializes c with std::move(q.c) as the first argument and a as the second argument, and
initializes comp with std::move(q.comp).

22.6.5.4 Members

void push(const value_type& x);

Effects: As if by:
    c.push_back(x);
push_heap(c.begin(), c.end(), comp);

void push(value_type&& x);

Effects: As if by:
    c.push_back(std::move(x));
push_heap(c.begin(), c.end(), comp);

template<class... Args> void emplace(Args&&... args);

Effects: As if by:
    c.emplace_back(std::forward<Args>(args)...);
push_heap(c.begin(), c.end(), comp);

void pop();

Effects: As if by:
    pop_heap(c.begin(), c.end(), comp);
c.pop_back();

22.6.5.5 Specialized algorithms

template<class T, class Container, class Compare>
    void swap(priority_queue<T, Container, Compare>& x,
              priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));

Constraints: is_swappable_v<Container> is true and is_swappable_v<Compare> is true.

Effects: As if by x.swap(y).

22.6.6 Class template stack

22.6.6.1 General

Any sequence container supporting operations back(), push_back() and pop_back() can be used to instan-
tiate stack. In particular, vector (22.3.11), list (22.3.10) and deque (22.3.8) can be used.

22.6.6.2 Definition

namespace std {
    template<class T, class Container = deque<T>>
    class stack {
    public:
```
using value_type = typename Container::value_type;
using reference = typename Container::reference;
using const_reference = typename Container::const_reference;
using size_type = typename Container::size_type;
using container_type = Container;

protected:
    Container c;

public:
    stack() : stack(Container()) {}
    explicit stack(const Container&);
    explicit stack(Container&&);
    template<class Alloc> explicit stack(const Alloc&);
    template<class Alloc> stack(const Container&, const Alloc&);
    template<class Alloc> stack(Container&&, const Alloc&);
    template<class Alloc> stack(const stack&, const Alloc&);
    template<class Alloc> stack(stack&&, const Alloc&);

    [[nodiscard]] bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference top() { return c.back(); }
    const_reference top() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void push(value_type&& x) { c.push_back(std::move(x)); }
    template<class... Args>
        decltype(auto) emplace(Args&&... args) { return c.emplace_back(std::forward<Args>(args)...); }
    void pop() { c.pop_back(); }
    void swap(stack& s) noexcept(is_nothrow_swappable_v<Container>) { using std::swap; swap(c, s.c); }

    template<class Container>
        stack(Container) -> stack<typename Container::value_type, Container>;
    template<class Container, class Allocator>
        stack(Container, Allocator) -> stack<typename Container::value_type, Container>;
    template<class T, class Container, class Alloc>
        struct uses_allocator<stack<T, Container>, Alloc> : uses_allocator<Container, Alloc>::type { };
template<class Alloc> stack(container_type&& cont, const Alloc& a);

Effects: Initializes c with std::move(cont) as the first argument and a as the second argument.

template<class Alloc> stack(const stack& s, const Alloc& a);

Effects: Initializes c with s.c as the first argument and a as the second argument.

template<class Alloc> stack(stack&& s, const Alloc& a);

Effects: Initializes c with std::move(s.c) as the first argument and a as the second argument.

22.6.6.5 Operators [stack.ops]

template<class T, class Container>
bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);

Returns: x.c == y.c.

template<class T, class Container>
bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);

Returns: x.c != y.c.

template<class T, class Container>
bool operator<(const stack<T, Container>& x, const stack<T, Container>& y);

Returns: x.c < y.c.

template<class T, class Container>
bool operator>(const stack<T, Container>& x, const stack<T, Container>& y);

Returns: x.c > y.c.

template<class T, class Container>
bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);

Returns: x.c <= y.c.

template<class T, class Container>
bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);

Returns: x.c >= y.c.

template<class T, three_way_comparable Container>
compare_three_way_result_t<Container>
operator<=>(const stack<T, Container>& x, const stack<T, Container>& y);

Returns: x.c <=> y.c.

22.6.6.6 Specialized algorithms [stack.special]

template<class T, class Container>
void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));

Constraints: is_swappable_v<Container> is true.

Effects: As if by x.swap(y).

22.7 Views [views]

22.7.1 General [views.general]

The header <span> defines the view span.

22.7.2 Header <span> synopsis [span.syn]

namespace std {
    // constants
    inline constexpr size_t dynamic_extent = numeric_limits<size_t>::max();

    // 22.7.3, class template span
    template<class ElementType, size_t Extent = dynamic_extent>
    class span;
template<class ElementType, size_t Extent>
inline constexpr bool ranges::enable_view<span<ElementType, Extent>> =
Extent == 0 || Extent == dynamic_extent;

template<class ElementType, size_t Extent>
inline constexpr bool ranges::enable_borrowed_range<span<ElementType, Extent>> = true;

// 22.7.3.8, views of object representation
-template<class ElementType, size_t Extent>
span<const byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent>
  as_bytes(span<ElementType, Extent> s) noexcept;

-template<class ElementType, size_t Extent>
span<byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent>
  as_writable_bytes(span<ElementType, Extent> s) noexcept;
}

// 22.7.3 Class template span
22.7.3.1 Overview
1 A span is a view over a contiguous sequence of objects, the storage of which is owned by some other object.
2 All member functions of span have constant time complexity.

namespace std {
  template<class ElementType, size_t Extent = dynamic_extent>
  class span {
  public:
    // constants and types
    using element_type = ElementType;
    using value_type = remove_cv_t<ElementType>;
    using size_type = size_t;
    using difference_type = ptrdiff_t;
    using pointer = element_type*;
    using const_pointer = const element_type*;
    using reference = element_type&;
    using const_reference = const element_type&;
    using iterator = implementation-defined; // see 22.7.3.7
    using reverse_iterator = std::reverse_iterator<iterator>;
    static constexpr size_type extent = Extent;

    // 22.7.3.2, constructors, copy, and assignment
    constexpr span() noexcept;
    template<class It>
      constexpr explicit(extent != dynamic_extent) span(It first, size_type count);
    template<class It, class End>
      constexpr explicit(extent != dynamic_extent) span(It first, End last);
    template<
      
    template<class T, size_t N>
      constexpr span(array<T, N>& arr) noexcept;
    template<class T, size_t N>
      constexpr span(const array<T, N>& arr) noexcept;
    template<class R>
      constexpr explicit(extent != dynamic_extent) span(R&& r);
    constexpr span(const span& other) noexcept = default;
    template<class OtherElementType, size_t OtherExtent>
      constexpr explicit(see below) span(const span<OtherElementType, OtherExtent>& s) noexcept;

    ~span() noexcept = default;
    constexpr span& operator=(const span& other) noexcept = default;

    // 22.7.3.4, subviews
    template<size_t Count>
      constexpr span<element_type, Count> first() const;
template<
  size_t Count>
constexpr span<element_type, Count> last() const;

template<
  size_t Offset, size_t Count = dynamic_extent>
constexpr span<element_type, see below> subspan() const;

constexpr span<element_type, dynamic_extent> first(size_type count) const;
constexpr span<element_type, dynamic_extent> last(size_type count) const;
constexpr span<element_type, dynamic_extent> subspan(
  size_type offset, size_type count = dynamic_extent) const;

// 22.7.3.5, observers
constexpr size_type size() const noexcept;
constexpr size_type size_bytes() const noexcept;
[[nodiscard]] constexpr bool empty() const noexcept;

// 22.7.3.6, element access
constexpr reference operator[](size_type idx) const;
constexpr reference front() const;
constexpr reference back() const;
constexpr pointer data() const noexcept;

// 22.7.3.7, iterator support
constexpr iterator begin() const noexcept;
constexpr iterator end() const noexcept;
constexpr reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator rend() const noexcept;

private:
  pointer data_; // exposition only
  size_type size_; // exposition only
};

template<class It, class EndOrSize>
span<It, EndOrSize> -> span<remove_reference_t<iter_reference_t<It>>>

3 ElementType is required to be a complete object type that is not an abstract class type.

22.7.3.2 Constructors, copy, and assignment [span.cons]

constexpr span() noexcept;

1 Constraints: Extent == dynamic_extent || Extent == 0 is true.

2 Postconditions: size() == 0 && data() == nullptr.

template<class It>
constexpr explicit(extent != dynamic_extent) span<It first, size_type count>;

3 Constraints: Let U be remove_reference_t<iter_reference_t<It>>.

(3.1) It satisfies contiguous_iterator.

(3.2) is_convertible_v<U(*[]), element_type(*)[]> is true.

[Note 1: The intent is to allow only qualification conversions of the iterator reference type to element_type.
  —end note]

4 Preconditions:

(4.1) [first, first + count) is a valid range.
(4.2) — It models contiguous_iterator.
(4.3) — If extent is not equal to dynamic_extent, then count is equal to extent.

Effects: Initializes data_ with to_address(first) and size_ with count.

Throws: Nothing.

template<class It, class End>
constexpr explicit(extent != dynamic_extent) span(It first, End last);

Constraints: Let U be remove_reference_t<iter_reference_t<It>>.
(7.1) — is_convertible_v<U(*)[], element_type(*)[]> is true.
      [Note 2: The intent is to allow only qualification conversions of the iterator reference type to element_type.]
      — end note]
(7.2) — It satisfies contiguous_iterator.
(7.3) — End satisfies sized_sentinel_for<It>.
(7.4) — is_convertible_v<End, size_t> is false.

Preconditions:
(8.1) — If extent is not equal to dynamic_extent, then last - first is equal to extent.
(8.2) — [first, last) is a valid range.
(8.3) — It models contiguous_iterator.
(8.4) — End models sized_sentinel_for<It>.

Effects: Initializes data_ with to_address(first) and size_ with last - first.

Throws: When and what last - first throws.

template<size_t N> constexpr span(type_identity_t<element_type> (&arr)[N]) noexcept;
template<class T, size_t N> constexpr span(array<T, N>& arr) noexcept;
template<class T, size_t N> constexpr span(const array<T, N>& arr) noexcept;

Constraints: Let U be remove_pointer_t<decltype(data(arr))>.
(11.1) — extent == dynamic_extent || N == extent is true, and
(11.2) — is_convertible_v<U(*)[], element_type(*)[]> is true.
      [Note 3: The intent is to allow only qualification conversions of the array element type to element_type.]
      — end note]

Effects: Constructs a span that is a view over the supplied array.
[Note 4: type_identity_t affects class template argument deduction. — end note]

Postconditions: size() == N && data() == data(arr) is true.

template<class R> constexpr explicit(extent != dynamic_extent) span(R&& r);

Constraints: Let U be remove_reference_t<ranges::range_reference_t<R>>.
(14.1) — R satisfies ranges::contiguous_range and ranges::sized_range.
(14.2) — Either R satisfies ranges::borrowed_range or is_const_v<element_type> is true.
(14.3) — remove_cvref_t<R> is not a specialization of span.
(14.4) — remove_cvref_t<R> is not a specialization of array.
(14.5) — is_array_v<remove_cvref_t<R>> is false.
(14.6) — is_convertible_v<U(*)[], element_type(*)[]> is true.
      [Note 5: The intent is to allow only qualification conversions of the range reference type to element_type.]
      — end note]

Preconditions:
(15.1) — If extent is not equal to dynamic_extent, then ranges::size(r) is equal to extent.
(15.2) — R models ranges::contiguous_range and ranges::sized_range.
(15.3) — If is_const_v<element_type> is false, R models ranges::borrowed_range.
Effects: Initializes data_ with ranges::data(r) and size_ with ranges::size(r).

Throws: What and when ranges::data(r) and ranges::size(r) throw.

16 constexpr span(const span& other) noexcept = default;

17 Postconditions: other.size() == size() && other.data() == data().

18 template<class OtherElementType, size_t OtherExtent>
19 constexpr explicit(see below) span(const span<OtherElementType, OtherExtent>& s) noexcept;

20 Constraints:
(19.1) extent == dynamic_extent || OtherExtent == dynamic_extent || extent == OtherExtent is true, and
(19.2) is_convertible_v<OtherElementType(*[]), element_type(*[])> is true.
[Note 6: The intent is to allow only qualification conversions of the OtherElementType to element_type. —end note]

21 Preconditions: If extent is not equal to dynamic_extent, then s.size() is equal to extent.

22 Effects: Constructs a span that is a view over the range [s.data(), s.data() + s.size()).

23 Postconditions: size() == s.size() && data() == s.data().

24 Remarks: The expression inside explicit is equivalent to:
    extent != dynamic_extent && OtherExtent == dynamic_extent

constexpr span& operator=(const span& other) noexcept = default;

25 Postconditions: size() == other.size() && data() == other.data().

22.7.3.3 Deduction guides

26 template<class It, class EndOrSize>
27 span(It, EndOrSize) -> span<remove_reference_t<iter_reference_t<It>>>;

28 Constraints: It satisfies contiguous_iterator.

29 template<class R>
30 span(R&&) -> span<remove_reference_t<ranges::range_reference_t<R>>>;

31 Constraints: R satisfies ranges::contiguous_range.

22.7.3.4 Subviews

32 template<size_t Count> constexpr span<element_type, Count> first() const;

33 Mandates: Count <= Extent is true.

34 Preconditions: Count <= size() is true.

35 Effects: Equivalent to: return R{data(), Count}; where R is the return type.

36 template<size_t Count> constexpr span<element_type, Count> last() const;

37 Mandates: Count <= Extent is true.

38 Preconditions: Count <= size() is true.

39 Effects: Equivalent to: return R{data() + (size() - Count), Count}; where R is the return type.

40 template<size_t Offset, size_t Count = dynamic_extent>
41 constexpr span<element_type, see below> subspan() const;

42 Mandates:
    Offset <= Extent && (Count == dynamic_extent || Count <= Extent - Offset)
    is true.

43 Preconditions:
    Offset <= size() && (Count == dynamic_extent || Count <= size() - Offset)
    is true.
Effects: Equivalent to:
```
return span<ElementType, see below>(
data() + Offset, Count != dynamic_extent ? Count : size() - Offset);
```

Remarks: The second template argument of the returned span type is:
```
Count != dynamic_extent ? Count
  : (Extent != dynamic_extent ? Extent - Offset
     : dynamic_extent)
```

```cpp
constexpr span<element_type, dynamic_extent> first(size_type count) const;
```

Preconditions: count <= size() is true.

Effects: Equivalent to: return {data(), count};

```cpp
constexpr span<element_type, dynamic_extent> last(size_type count) const;
```

Preconditions: count <= size() is true.

Effects: Equivalent to: return {data() + (size() - count), count};

```cpp
constexpr span<element_type, dynamic_extent> subspan(
    size_type offset, size_type count = dynamic_extent) const;
```

Preconditions:
```
offset <= size() && (count == dynamic_extent || count <= size() - offset)
```
is true.

Effects: Equivalent to:
```
return {data() + offset, count == dynamic_extent ? size() - offset : count};
```

22.7.3.5 Observers

```cpp
constexpr size_type size() const noexcept;
```

Effects: Equivalent to: return size_

```cpp
constexpr size_type size_bytes() const noexcept;
```

Effects: Equivalent to: return size() * sizeof(element_type);

```cpp
[[nodiscard]] constexpr bool empty() const noexcept;
```

Effects: Equivalent to: return size() == 0;

22.7.3.6 Element access

```cpp
constexpr reference operator[](size_type idx) const;
```

Preconditions: idx < size() is true.

Effects: Equivalent to: return *(data() + idx);

```cpp
constexpr reference front() const;
```

Preconditions: empty() is false.

Effects: Equivalent to: return *data();

```cpp
constexpr reference back() const;
```

Preconditions: empty() is false.

Effects: Equivalent to: return *(data() + (size() - 1));
22.7.3.7 Iterator support

using iterator = implementation-defined;

1 The type models contiguous_iterator (23.3.4.14), meets the Cpp17RandomAccessIterator requirements (23.3.5.7), and meets the requirements for constexpr iterators (23.3.1), whose value type is value_type and whose reference type is reference.

2 All requirements on container iterators (22.2) apply to span::iterator as well.

constexpr iterator begin() const noexcept;

1 Returns: An iterator referring to the first element in the span. If empty() is true, then it returns the same value as end().

constexpr iterator end() const noexcept;

2 Returns: An iterator which is the past-the-end value.

constexpr reverse_iterator rbegin() const noexcept;

3 Effects: Equivalent to: return reverse_iterator(end());

constexpr reverse_iterator rend() const noexcept;

4 Effects: Equivalent to: return reverse_iterator(begin());

22.7.3.8 Views of object representation

template<class ElementType, size_t Extent>
span<const byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent>
as_bytes(span<ElementType, Extent> s) noexcept;

1 Effects: Equivalent to: return R{reinterpret_cast<const byte*>(s.data()), s.size_bytes()}; where R is the return type.

template<class ElementType, size_t Extent>
span<byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent>
as_writable_bytes(span<ElementType, Extent> s) noexcept;

2 Constraints: is_const_v<ElementType> is false.

3 Effects: Equivalent to: return R{reinterpret_cast<byte*>(s.data()), s.size_bytes()}; where R is the return type.
23 Iterators library

23.1 General

This Clause describes components that C++ programs may use to perform iterations over containers (Clause 22), streams (29.7), stream buffers (29.6), and other ranges (Clause 24).

The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 82.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.3</td>
<td>Iterator requirements &lt;iterator&gt;</td>
</tr>
<tr>
<td>23.4</td>
<td>Iterator primitives</td>
</tr>
<tr>
<td>23.5</td>
<td>Iterator adaptors</td>
</tr>
<tr>
<td>23.6</td>
<td>Stream iterators</td>
</tr>
<tr>
<td>23.7</td>
<td>Range access</td>
</tr>
</tbody>
</table>

23.2 Header <iterator> synopsis

```
#include <compare> // see 17.11.1
#include <concepts> // see 18.3

namespace std {
    template<class T> using with-reference = T&; // exposition only
    template<class T> concept can-reference // exposition only
        = requires { typename with-reference<T> ; };
    template<class T> concept dereferenceable // exposition only
        = requires(T& t) { *(t) -> can-reference; // not required to be equality-preserving };

    // 23.3.2, associated types
    // 23.3.2.1, incrementable traits
    template<class> struct incrementable_traits;
    template<class T>
        using iter_difference_t = see below;

    // 23.3.2.2, indirectly readable traits
    template<class> struct indirectly_readable_traits;
    template<class T>
        using iter_value_t = see below;

    // 23.3.2.3, iterator traits
    template<class I> struct iterator_traits;
    template<class T> requires is_object_v<T> struct iterator_traits<T*>;

    template<dereferenceable T>
        using iter_reference_t = decltype(*declval<T&>());

    namespace ranges {
        // 23.3.3, customization point objects
        inline namespace unspecified {
            // 23.3.3.1, ranges::iter_move
            inline constexpr unspecified iter_move = unspecified;
        }
    }
```
// 23.3.3.2, ranges::iter_swap
inline constexpr unspecified iter_swap = unspecified;
}

template<
dereferenceable T>
requires requires(T& t) {
    { ranges::iter_move(t) } -> can-reference;
}
using iter_rvalue_reference_t
    = decltype(ranges::iter_move(declval<T&>()));

// 23.3.4, iterator concepts
// 23.3.4.2, concept indirectly_readable
template<class In>
    concept indirectly_readable = see below;

template<indirectly_readable T>
    using iter_common_reference_t
        = common_reference_t<iter_reference_t<T>, iter_value_t<T>&>;

// 23.3.4.3, concept indirectly_writable
template<class Out, class T>
    concept indirectly_writable = see below;

// 23.3.4.4, concept weakly_incrementable
template<class I>
    concept weakly_incrementable = see below;

// 23.3.4.5, concept incrementable
template<class I>
    concept incrementable = see below;

// 23.3.4.6, concept input_or_output_iterator
template<class I>
    concept input_or_output_iterator = see below;

// 23.3.4.7, concept sentinel_for
template<class S, class I>
    concept sentinel_for = see below;

// 23.3.4.8, concept sized_sentinel_for
template<class S, class I>
    inline constexpr bool disable_sized_sentinel_for = false;

    template<class S, class I>
        concept sized_sentinel_for = see below;

// 23.3.4.9, concept input_iterator
template<class I>
    concept input_iterator = see below;

// 23.3.4.10, concept output_iterator
template<class I, class T>
    concept output_iterator = see below;

// 23.3.4.11, concept forward_iterator
template<class I>
    concept forward_iterator = see below;

// 23.3.4.12, concept bidirectional_iterator
template<class I>
    concept bidirectional_iterator = see below;
// 23.3.4.13, concept random_access_iterator
template<class I>
concept random_access_iterator = see below;

// 23.3.4.14, concept contiguous_iterator
template<class I>
concept contiguous_iterator = see below;

// 23.3.6, indirect callable requirements
// 23.3.6.2, indirect callables
template<class F, class I>
concept indirectly_unary_invocable = see below;

template<class F, class I>
concept indirectly_regular_unary_invocable = see below;

template<class F, class I>
concept indirect_unary_predicate = see below;

template<class F, class I1, class I2>
concept indirect_binary_predicate = see below;

template<class F, class I1, class I2 = I1>
concept indirect_equivalence_relation = see below;

template<class F, class I1, class I2 = I1>
concept indirect_strict_weak_order = see below;

template<class F, class... Is>
requires (indirectly_readable<Is> && ...) && invocable<F, iter_reference_t<Is>...>
using indirect_result_t = invoke_result_t<F, iter_reference_t<Is>...>;

// 23.3.6.3, projected
template<indirectly_readable I, indirectly_regular_unary_invocable<I> Proj>
struct projected;

template<weakly_incrementable I, class Proj>
struct incrementable_traits<projected<I, Proj>>;

// 23.3.7, common algorithm requirements
// 23.3.7.2, concept indirectly_movable
template<class In, class Out>
concept indirectly_movable = see below;

template<class In, class Out>
concept indirectly_movable_storable = see below;

// 23.3.7.3, concept indirectly_copyable
template<class In, class Out>
concept indirectly_copyable = see below;

template<class In, class Out>
concept indirectly_copyable_storable = see below;

// 23.3.7.4, concept indirectly_swappable
template<class I1, class I2 = I1>
concept indirectly_swappable = see below;

// 23.3.7.5, concept indirectly_comparable
template<class I1, class I2, class R, class P1 = identity, class P2 = identity>
concept indirectly_comparable = see below;
// 23.3.7.6, concept permutable
template<class I>
concept permutable = see below;

// 23.3.7.7, concept mergeable
template<class I1, class I2, class Out,
class R = ranges::less, class P1 = identity, class P2 = identity>
concept mergeable = see below;

// 23.3.7.8, concept sortable
template<class I, class R = ranges::less, class P = identity>
concept sortable = see below;

// 23.4, primitives
// 23.4.2, iterator tags
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag: public input_iterator_tag { };
struct bidirectional_iterator_tag: public forward_iterator_tag { };
struct random_access_iterator_tag: public bidirectional_iterator_tag { };
struct contiguous_iterator_tag: public random_access_iterator_tag { };

// 23.4.3, iterator operations
template<class InputIterator, class Distance>
constexpr void
advance(InputIterator& i, Distance n);
template<class InputIterator>
constexpr typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
template<class InputIterator>
constexpr InputIterator
next(InputIterator x,
typename iterator_traits<InputIterator>::difference_type n = 1);
template<class BidirectionalIterator>
constexpr BidirectionalIterator
prev(BidirectionalIterator x,
typename iterator_traits<BidirectionalIterator>::difference_type n = 1);

// 23.4.4, range iterator operations
namespace ranges {
// 23.4.4.2, ranges::advance
template<input_or_output_iterator I>
constexpr void advance(I& i, iter_difference_t<I> n);
template<input_or_output_iterator I, sentinel_for<I> S>
constexpr void advance(I& i, S bound);
template<input_or_output_iterator I, sentinel_for<I> S>
constexpr iter_difference_t<I> advance(I& i, iter_difference_t<I> n, S bound);

// 23.4.4.3, ranges::distance
template<input_or_output_iterator I, sentinel_for<I> S>
constexpr iter_difference_t<I> distance(I first, S last);
template<range R>
constexpr range_difference_t<R> distance(R&& r);

// 23.4.4.4, ranges::next
template<input_or_output_iterator I>
constexpr I next(I x);
template<input_or_output_iterator I>
constexpr I next(I x, iter_difference_t<I> n);
template<input_or_output_iterator I, sentinel_for<I> S>
constexpr I next(I x, S bound);
template<input_or_output_iterator I, sentinel_for<I> S>
constexpr I next(I x, iter_difference_t<I> n, S bound);
// 23.4.4.5, ranges::prev
template<bidirectional_iterator I>
constexpr I prev(I x);

// 23.5, predefined iterators and sentinels
// 23.5.1, reverse iterators

template<class Iterator> class reverse_iterator;

template<class Iterator1, class Iterator2>
constexpr bool operator==(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator!=(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator<(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator>(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator<=(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator>=(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2, three_way_comparable_with<Iterator1> Iterator2>
constexpr compare_three_way_result_t<Iterator1, Iterator2> operator<=>(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr auto operator-(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());

template<class Iterator>
constexpr reverse_iterator<Iterator> operator+(iter_difference_t<Iterator> n, const reverse_iterator<Iterator>& x);

// 23.5.2, insert iterators

template<class Container> class back_insert_iterator;

template<class Container>
constexpr back_insert_iterator<Container> back_inserter(Container& x);

§ 23.2
template<class Container> class front_insert_iterator;

template<class Container>
  constexpr front_insert_iterator<Container> front_inserter(Container& x);

template<class Container> class insert_iterator;

template<class Container>
  constexpr insert_iterator<Container> inserter(Container& x, ranges::iterator_t<Container> i);

// 23.5.3, move iterators and sentinels

template<class Iterator> class move_iterator;

template<class Iterator1, class Iterator2>
  constexpr bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
  constexpr bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
  constexpr bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
  constexpr bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
  constexpr bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
  constexpr compare_three_way_result_t<Iterator1, Iterator2> operator<=>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
  constexpr auto operator-(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y)
    -> decltype(x.base() - y.base());

template<class Iterator>
  constexpr move_iterator<Iterator> operator+(iter_difference_t<Iterator> n, const move_iterator<Iterator>& x);

template<class Iterator>
  constexpr move_iterator<Iterator> make_move_iterator(Iterator i);

template<semiregular S> class move_sentinel;

// 23.5.4, common iterators

template<input_or_output_iterator I, sentinel_for<I> S>
  requires (!same_as<I, S> && copyable<I>)
  class common_iterator;

template<class I, class S>
  struct incrementable_traits<common_iterator<I, S>>;

template<input_iterator I, class S>
  struct iterator_traits<common_iterator<I, S>>;

// 23.5.5, default sentinel

struct default_sentinel_t;
inline constexpr default_sentinel_t default_sentinel{};

// 23.5.6, counted iterators

template<input_or_output_iterator I> class counted_iterator;
template<class I>
    struct incrementable_traits<counted_iterator<I>>;

template<input_iterator I>
    struct iterator_traits<counted_iterator<I>>;

// 23.5.7, unreachable sentinel
struct unreachable_sentinel_t;
inline constexpr unreachable_sentinel_t unreachable_sentinel{};

// 23.6, stream iterators

template<class T, class charT = char, class traits = char_traits<charT>,
class Distance = ptrdiff_t>
    class istream_iterator;

template<class T, class charT, class traits, class Distance>
    bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
                    const istream_iterator<T,charT,traits,Distance>& y);

// 23.6.3, range access

template<class C> constexpr auto begin(C& c) -> decltype(c.begin());
template<class C> constexpr auto begin(const C& c) -> decltype(c.begin());

// 23.7, range access

template<class C> constexpr auto end(C& c) -> decltype(c.end());
template<class C> constexpr auto end(const C& c) -> decltype(c.end());

// 23.7.1, range access

template<class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
template<class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;

// 23.7.2, range access

template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
    -> decltype(std::begin(c));
template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
    -> decltype(std::end(c));

template<class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin());
template<class C> constexpr auto rbegin(const C& c) -> decltype(c.rbegin());

// 23.7.3, range access

template<class C> constexpr auto size(const C& c) -> decltype(c.size());
template<class T, size_t N> constexpr size_t size(const T (&array)[N]) noexcept;

// 23.7.4, range access

template<class C> constexpr auto empty(const C& c) -> decltype(c.empty());
template<class T, size_t N> constexpr bool empty(const T (&array)[N]) noexcept;

// 23.7.5, range access

template<class C> constexpr auto data(C& c) -> decltype(c.data());
template<class C> constexpr auto data(const C& c) -> decltype(c.data());

// 23.7.6, range access

template<class T, size_t N> constexpr T* data(T (&array)[N]) noexcept;

// 23.7.7, range access

template<class C> constexpr auto begin(C& c) -> decltype(c.begin());
template<class C> constexpr auto begin(const C& c) -> decltype(c.begin());

// 23.7.8, range access

template<class C> constexpr auto end(C& c) -> decltype(c.end());
template<class C> constexpr auto end(const C& c) -> decltype(c.end());

// 23.7.9, range access

template<class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;

// 23.7.10, range access

template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
    -> decltype(std::begin(c));
template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
    -> decltype(std::end(c));

template<class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin());
template<class C> constexpr auto rbegin(const C& c) -> decltype(c.rbegin());
template<class E> constexpr const E* data(initializer_list<E> il) noexcept;

23.3 Iterator requirements

23.3.1 In general

Iterators are a generalization of pointers that allow a C++ program to work with different data structures (for example, containers and ranges) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. An input iterator \( i \) supports the expression \( *i \), resulting in a value of some object type \( T \), called the value type of the iterator. An output iterator \( i \) has a non-empty set of types that are indirectly_writable to the iterator; for each such type \( T \), the expression \( *i = o \) is valid where \( o \) is a value of type \( T \). For every iterator type \( X \), there is a corresponding signed integer-like type (23.3.4.4) called the difference type of the iterator.

Since iterators are an abstraction of pointers, their semantics are a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This document defines six categories of iterators, according to the operations defined on them: input iterators, output iterators, forward iterators, bidirectional iterators, random access iterators, and contiguous iterators, as shown in Table 83.

Table 83: Relations among iterator categories

<table>
<thead>
<tr>
<th>Contiguous</th>
<th>Random Access</th>
<th>Bidirectional</th>
<th>Forward</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
</table>

The six categories of iterators correspond to the iterator concepts:

1. input_iterator (23.3.4.9),
2. output_iterator (23.3.4.10),
3. forward_iterator (23.3.4.11),
4. bidirectional_iterator (23.3.4.12),
5. random_access_iterator (23.3.4.13), and
6. contiguous_iterator (23.3.4.14), respectively. The generic term iterator refers to any type that models the input_or_output_iterator concept (23.3.4.6).

Forward iterators meet all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also meet all the requirements of forward iterators and can be used whenever a forward iterator is specified; Random access iterators also meet all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified; Contiguous iterators also meet all the requirements of random access iterators and can be used whenever a random access iterator is specified.

Iterators that further meet the requirements of output iterators are called mutable iterators. Nonmutable iterators are referred to as constant iterators.

In addition to the requirements in this subclause, the nested typedef-names specified in 23.3.2.3 shall be provided for the iterator type.

[Note 1: Either the iterator type must provide the typedef-names directly (in which case iterator_traits pick them up automatically), or an iterator_traits specialization must provide them. — end note]

Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. Such a value is called a past-the-end value. Values of an iterator \( i \) for which the expression \( *i \) is defined are called dereferenceable. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any sequence. Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular value to an iterator that holds a singular value, and, for iterators that meet the Cpp17DefaultConstructible requirements, using a value-initialized iterator as the source of a copy or move operation.
© ISO/IEC

[Note 2: This guarantee is not offered for default-initialization, although the distinction only matters for types with trivial default constructors such as pointers or aggregates holding pointers. — end note]

In these cases the singular value is overwritten the same way as any other value. Dereferenceable values are always non-singular.

Most of the library’s algorithmic templates that operate on data structures have interfaces that use ranges. A range is an iterator and a sentinel that designate the beginning and end of the computation, or an iterator and a count that designate the beginning and the number of elements to which the computation is to be applied.\(^{234}\)

An iterator and a sentinel denoting a range are comparable. A range \([i, s)\) is empty if \(i == s\); otherwise, \([i, s)\) refers to the elements in the data structure starting with the element pointed to by \(i\) and up to but not including the element, if any, pointed to by the first iterator \(j\) such that \(j == s\).

A sentinel \(s\) is called reachable from an iterator \(i\) if and only if there is a finite sequence of applications of the expression \(++i\) that makes \(i == s\). If \(s\) is reachable from \(i\), \([i, s)\) designates a valid range.

A counted range \(i + [0, n)\) is empty if \(n == 0\); otherwise, \(i + [0, n)\) refers to the \(n\) elements in the data structure starting with the element pointed to by \(i\) and up to but not including the element, if any, pointed to by the result of \(n\) applications of \(++i\). A counted range \(i + [0, n)\) is valid if and only if \(n == 0\); or \(n\) is positive, \(i\) is dereferenceable, and \(++i + [0, --n)\) is valid.

The result of the application of library functions to invalid ranges is undefined.

All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized). Therefore, requirement tables and concept definitions for the iterators do not specify complexity.

Destruction of a non-forward iterator may invalidate pointers and references previously obtained from that iterator.

An invalid iterator is an iterator that may be singular.\(^{235}\)

Iterators are called constexpr iterators if all operations provided to meet iterator category requirements are constexpr functions.

[Note 3: For example, the types “pointer to int” and reverse_iterator<int*> are constexpr iterators. — end note]

\section{23.3.2 Associated types}

\subsection{23.3.2.1 Incrementable traits}

To implement algorithms only in terms of incrementable types, it is often necessary to determine the difference type that corresponds to a particular incrementable type. Accordingly, it is required that if \(WI\) is the name of a type that models the weakly_incrementable concept (23.3.4.4), the type

\[\text{iter_difference_t<}\text{\langle}WI\text{\rangle}\]

be defined as the incrementable type’s difference type.

\begin{verbatim}
namespace std {
  template<class> struct incrementable_traits { }

  template<class T>
  requires is_object_v<T>
  struct incrementable_traits<T> { 
    using difference_type = ptrdiff_t;
  };

  template<class I>
  struct incrementable_traits<const I> : incrementable_traits<I> { };

  template<class T>
  requires requires { typename T::difference_type; }

\end{verbatim}
struct incrementable_traits<T> {
    using difference_type = typename T::difference_type;
};

template<class T>
    requires (!requires { typename T::difference_type; } &&
    requires(const T& a, const T& b) { { a - b } -> integral; })
struct incrementable_traits<T> {
    using difference_type = make_signed_t<decltype(declval<T>() - declval<T>())>;
};

template<class T>
    using iter_difference_t = see below;

Let \( R_I \) be remove_cvref_t<I>. The type \( \text{iter\_difference\_t}<I> \) denotes

(2.1) \( \text{incrementable\_traits}<R_I>::\text{difference\_type} \) if iterator_traits<\( R_I \)> names a specialization generated from the primary template, and

(2.2) \( \text{iterator\_traits}<R_I>::\text{difference\_type} \) otherwise.

Users may specialize \text{incrementable\_traits} on program-defined types.

23.3.2.2 Indirectly readable traits

To implement algorithms only in terms of indirectly readable types, it is often necessary to determine the value type that corresponds to a particular indirectly readable type. Accordingly, it is required that if \( R \) is the name of a type that models the \textit{indirectly\_readable} concept (23.3.4.2), the type

\( \text{iter\_value\_t}<R> \)

be defined as the indirectly readable type's value type.

template<class> struct cond-value-type { }; // exposition only
template<class T>
    requires is_object_v<T>
struct cond-value-type<T> {
    using value_type = remove_cv_t<T>;
};

template<class T>
struct indirectly_readable_traits { };

template<class T>
struct indirectly_readable_traits<T*> : cond-value-type<T> { };

template<class I>
    requires is_array_v<I>
struct indirectly_readable_traits<I> { 
    using value_type = remove_cv_t<remove_extent_t<I>>;
};

template<class I>
struct indirectly_readable_traits<const I> : indirectly_readable_traits<I> { };

template<class T>
    requires requires { typename T::value_type; }
struct indirectly_readable_traits<T> : cond-value-type<typename T::value_type> { };

template<class T>
    requires requires { typename T::element_type; }
struct indirectly_readable_traits<T> : cond-value-type<typename T::element_type> { };

§ 23.3.2.2 931
template<class T> using iter_value_t = see below;

2 Let \( R_I \) be remove_cvref_t\( I \). The type \( \text{iter_value_t}\langle R_I \rangle \) denotes

\[(2.1) \quad \text{indirectly_readable_traits}\langle R_I \rangle::\text{value_type} \quad \text{if} \quad \text{iterator_traits}\langle R_I \rangle \quad \text{names a specialization} \quad \text{generated from the primary template,} \quad \text{and} \]

\[(2.2) \quad \text{iterator_traits}\langle R_I \rangle::\text{value_type} \quad \text{otherwise}. \]

3 Class template \text{indirectly_readable_traits} may be specialized on program-defined types.

[Note 1: Some legacy output iterators define a nested type named \text{value_type} that is an alias for \text{void}. These types are not \text{indirectly_readable} and have no associated value types. — end note]

[Note 2: Smart pointers like \text{shared_ptr<int>} are \text{indirectly_readable} and have an associated value type, but a smart pointer like \text{shared_ptr<void>} is not \text{indirectly_readable} and has no associated value type. — end note]

23.3.2.3 Iterator traits

To implement algorithms only in terms of iterators, it is sometimes necessary to determine the iterator category that corresponds to a particular iterator type. Accordingly, it is required that if \( I \) is the type of an iterator, the type

\[
\text{iterator_traits}\langle I \rangle::\text{iterator_category}
\]

be defined as the iterator’s iterator category. In addition, the types

\[
\begin{align*}
\text{iterator_traits}\langle I \rangle::\text{pointer} \\
\text{iterator_traits}\langle I \rangle::\text{reference}
\end{align*}
\]

shall be defined as the iterator’s pointer and reference types; that is, for an iterator object \( a \) of class type, the same type as \( \text{decltype}(a.\text{operator}\rightarrow()) \) and \( \text{decltype}(\ast a) \), respectively. The type \( \text{iterator_traits}\langle I \rangle::\text{pointer} \) shall be \text{void} for an iterator of class type \( I \) that does not support \( \text{operator}\rightarrow\). Additionally, in the case of an output iterator, the types

\[
\begin{align*}
\text{iterator_traits}\langle I \rangle::\text{value_type} \\
\text{iterator_traits}\langle I \rangle::\text{difference_type} \\
\text{iterator_traits}\langle I \rangle::\text{reference}
\end{align*}
\]

may be defined as \text{void}.

2 The definitions in this subclause make use of the following exposition-only concepts:

template<class I>
concept cpp17_iterator =
    copyable\langle I \rangle &\& requires(I i) {} {
    \{ *i \} -> can-reference; \\
    \{ ++i \} -> same_as\langle I&\rangle; \\
    \{ *i++ \} -> can-reference;
    };

template<class I>
concept cpp17_input_iterator =
    cpp17_iterator\langle I \rangle &\& equality_comparable\langle I \rangle &\& requires(I i) {} {
    typename incrementable_traits\langle I \rangle::\text{difference_type}; \\
    typename indirectly_readable_traits\langle I \rangle::\text{value_type}; \\
    typename common_reference_t\langle iter_reference_t\langle I \rangle&\&, \\
                                typename indirectly_readable_traits\langle I \rangle::\text{value_type}&\rangle; \\
    typename common_reference_t\langle decltype(*i++)&\&, \\
                                typename indirectly_readable_traits\langle I \rangle::\text{value_type}&\rangle; \\
    requires signed_integral\langle typename incrementable_traits\langle I \rangle::\text{difference_type}\rangle;
    };

template<class I>
concept cpp17_forward_iterator =
    cpp17_input_iterator\langle I \rangle &\& constructible_from\langle I \rangle &\&
    is_lvalue_reference_v\langle iter_reference_t\langle I \rangle\rangle &\&
    same_as\langle remove_cvref_t\langle iter_reference_t\langle I \rangle\rangle, \\
             typename indirectly_readable_traits\langle I \rangle::\text{value_type}\rangle &\&
    requires(I i) {} {
    \{ i++ \} -> convertible_to\langle const I&\rangle; \\
    \{ *i++ \} -> same_as\langle iter_reference_t\langle I \rangle\rangle;
    };

§ 23.3.2.3  932
§ 23.3.2.3

The members of a specialization `iterator_traits<I>` generated from the `iterator_traits` primary template are computed as follows:

1. If the `qualified-id` `I::pointer` is valid and denotes a type, then `pointer` names that type. Otherwise, if `decltype(declval<I&>().operator->())` is well-formed, then `pointer` names that type.
   
   Otherwise, `pointer` names `void`.

2. If the `qualified-id` `I::reference` is valid and denotes a type, then `reference` names that type. Otherwise, `reference` names `iter_reference_t<I>`.

3. Otherwise, if I satisfies the exposition-only concept `cpp17-input-iterator`, then `iterator_traits<I>` has the following publicly accessible members:

   ```cpp
   template<class I>
   concept cpp17-input-iterator =
   cpp17-forward-iterator<I> && requires(I i) {
     { --i } -> same_as<I&>;
     { i-- } -> convertible_to<const I&>;
     { *i-- } -> same_as<iter_reference_t<I>>;
   };
   ```

   ```cpp
   template<class I>
   concept cpp17-bidirectional-iterator =
   cpp17-forward-iterator<I> && requires(I i) {
     { --i } -> same_as<I&>;
     { i-- } -> convertible_to<const I&>;
     { *i-- } -> same_as<iter_reference_t<I>>;
   };
   ```

   ```cpp
   template<class I>
   concept cpp17-random-access-iterator =
   cpp17-bidirectional-iterator<I> && totally_ordered<I> && requires(I i, typename incrementable_traits<I>::difference_type n) {
     { i += n } -> same_as<I&>;
     { i -= n } -> same_as<I&>;
     { n + i } -> same_as<I>;
     { i + n } -> same_as<I>;
     { i - n } -> same_as<I>;
     { i - i } -> same_as<decltype(n)>;
     { i[n] } -> convertible_to<iter_reference_t<I>>;
   };
   ```

   The members of a specialization `iterator_traits<I>` generated from the `iterator_traits` primary template are computed as follows:

   - If I has valid (13.10.3) member types `difference_type`, `value_type`, `reference`, and `iterator_category`, then `iterator_traits<I>` has the following publicly accessible members:

     ```cpp
     using iterator_category = typename I::iterator_category;
     using value_type = typename I::value_type;
     using difference_type = typename I::difference_type;
     using pointer = see below;
     using reference = see below;
     ```

   - Otherwise, if I satisfies the exposition-only concept `cpp17-input-iterator`, `iterator_traits<I>` has the following publicly accessible members:

     ```cpp
     using iterator_category = output_iterator_tag;
     using value_type = void;
     using difference_type = see below;
     ```

§ 23.3.2.3
using pointer = void;
using reference = void;

If the qualified-id `incrementable_traits<T>::difference_type` is valid and denotes a type, then `difference_type` names that type; otherwise, it names `void`.

(3.4)

— Otherwise, `iterator_traits<T>` has no members by any of the above names.

4 Explicit or partial specializations of `iterator_traits` may have a member type `iterator_concept` that is used to indicate conformance to the iterator concepts (23.3.4).

5 `iterator_traits` is specialized for pointers as

```cpp
namespace std {
    template<class T>
    requires is_object_v<T>
    struct iterator_traits<T*> {
        using iterator_concept = contiguous_iterator_tag;
        using iterator_category = random_access_iterator_tag;
        using value_type = remove_cv_t<T>;
        using difference_type = ptrdiff_t;
        using pointer = T*;
        using reference = T&;
    };
}
```

6 [Example 1: To implement a generic `reverse` function, a C++ program can do the following:

```cpp
template<class BI>
void reverse(BI first, BI last) {
    typename iterator_traits<BI>::difference_type n =
        distance(first, last);
    --n;
    while(n > 0) {
        typename iterator_traits<BI>::value_type tmp = *first;
        *first++ = *--last;
        *last = tmp;
        n -= 2;
    }
}
```

— end example]

23.3.3 Customization point objects [iterator.cust]

23.3.3.1 ranges::iter_move [iterator.cust.move]

1 The name `ranges::iter_move` denotes a customization point object (16.3.3.3.6). The expression `ranges::iter_move(E)` for a subexpression `E` is expression-equivalent to:

(1.1) — `iter_move(E)`, if `E` has class or enumeration type and `iter_move(E)` is a well-formed expression when treated as an unevaluated operand, with overload resolution performed in a context that does not include a declaration of `ranges::iter_move` but does include the declaration

```cpp
void iter_move();
```

(1.2) — Otherwise, if the expression `*E` is well-formed:

(1.2.1) — if `*E` is an lvalue, `std::move(*E);`

(1.2.2) — otherwise, `*E`.

(1.3) — Otherwise, `ranges::iter_move(E)` is ill-formed.

[Note 1: This case can result in substitution failure when `ranges::iter_move(E)` appears in the immediate context of a template instantiation. — end note]

2 If `ranges::iter_move(E)` is not equal to `*E`, the program is ill-formed, no diagnostic required.

23.3.3.2 ranges::iter_swap [iterator.cust.swap]

1 The name `ranges::iter_swap` denotes a customization point object (16.3.3.3.6) that exchanges the values (18.4.9) denoted by its arguments.
Let \texttt{iter-exchange-move} be the exposition-only function:

\begin{verbatim}
template<class X, class Y>
constexpr iter_value_t<X> iter-exchange-move(X&& x, Y&& y)
  noexcept(noexcept(iter_value_t<X>(iter_move(x))) &&
  noexcept(*x = iter_move(y)));
\end{verbatim}

**Effects:** Equivalent to:

\begin{verbatim}
iter_value_t<X> old_value(iter_move(x));
*x = iter_move(y);
return old_value;
\end{verbatim}

The expression \texttt{ranges::iter_swap(E1, E2)} for subexpressions \(E1\) and \(E2\) is expression-equivalent to:

\begin{itemize}
  \item (void)\texttt{iter_swap(E1, E2)}, if either \(E1\) or \(E2\) has class or enumeration type and \texttt{iter_swap(E1, E2)} is a well-formed expression with overload resolution performed in a context that includes the declaration
  \begin{verbatim}
  template<class I1, class I2>
  void iter_swap(I1, I2) = delete;
  \end{verbatim}
  and does not include a declaration of \texttt{ranges::iter_swap}. If the function selected by overload resolution does not exchange the values denoted by \(E1\) and \(E2\), the program is ill-formed, no diagnostic required.
  \item Otherwise, if the types of \(E1\) and \(E2\) each model \texttt{indirectly_readable}, and if the reference types of \(E1\) and \(E2\) model \texttt{swappable_with} (18.4.9), then (void)(\(E1 = \texttt{iter-exchange-move}(E2, E1)\)), except that \(E1\) is evaluated only once.
  \item Otherwise, \texttt{ranges::iter_swap(E1, E2)} is ill-formed.
\end{itemize}

[Note 1: This case can result in substitution failure when \texttt{ranges::iter_swap(E1, E2)} appears in the immediate context of a template instantiation. — end note]

### 23.3.4 Iterator concepts

#### 23.3.4.1 General

For a type \(I\), let \texttt{ITER_TRAITS(I)} denote the type \(I\) if \texttt{iterator_traits<I>} names a specialization generated from the primary template. Otherwise, \texttt{ITER_TRAITS(I)} denotes \texttt{iterator_traits<I>}.

\begin{itemize}
  \item If the qualified-id \texttt{ITER_TRAITS(I)::iterator_concept} is valid and names a type, then \texttt{ITER_CONCEPT(I)} denotes that type.
  \item Otherwise, if the qualified-id \texttt{ITER_TRAITS(I)::iterator_category} is valid and names a type, then \texttt{ITER_CONCEPT(I)} denotes that type.
  \item Otherwise, \texttt{iterator_traits<I>} names a specialization generated from the primary template, then \texttt{ITER_CONCEPT(I)} denotes \texttt{random_access_iterator_tag}.
  \item Otherwise, \texttt{ITER_CONCEPT(I)} does not denote a type.
\end{itemize}

[Note 1: \texttt{ITER_TRAITS} enables independent syntactic determination of an iterator's category and concept. — end note]

[Example 1:

```cpp
struct I {
  using value_type = int;
  using difference_type = int;

  int operator*() const;
  I& operator++();
  I operator++(int);
  I& operator--();
  I operator--(int);

  bool operator==(I) const;
};
```

\texttt{iterator_traits<I>::iterator_concept} denotes \texttt{input_iterator_tag}, and \texttt{ITER_CONCEPT(I)} denotes \texttt{random_access_iterator_tag}. — end example]
23.3.4.2 Concept indirectly_readable

Types that are indirectly readable by applying operator* model the indirectly_readable concept, including pointers, smart pointers, and iterators.

```cpp
template<class In>
concept indirectly-readable-impl =
requires(const In in) {
    typename iter_value_t<In>;
    typename iter_reference_t<In>;
    typename iter_rvalue_reference_t<In>;
    { *in } -> same_as<iter_reference_t<In>>;
    { ranges::iter_move(in) } -> same_as<iter_rvalue_reference_t<In>>;
} &&
common_reference_with<iter_reference_t<In>&&, iter_value_t<In>&> &&
common_reference_with<iter_reference_t<In>&&, iter_rvalue_reference_t<In>&&> &&
common_reference_with<iter_rvalue_reference_t<In>&&, const iter_value_t<In>&>;

template<class In>
concept indirectly_readable = indirectly-readable-impl<remove_cvref_t<In>>;
```

Given a value i of type I, I models indirectly_readable only if the expression *i is equality-preserving.

[Note 1: The expression *i is indirectly required to be valid via the exposition-only dereferenceable concept (23.2). —end note]

23.3.4.3 Concept indirectly_writable

The indirectly_writable concept specifies the requirements for writing a value into an iterator’s referenced object.

```cpp
template<class Out, class T>
concept indirectly_writable =
requires(Out&& o, T&& t) {
    *o = std::forward<T>(t); // not required to be equality-preserving
    *std::forward<Out>(o) = std::forward<T>(t); // not required to be equality-preserving
    const_cast<const iter_reference_t<Out>&&>(*o) = std::forward<T>(t); // not required to be equality-preserving
    const_cast<const iter_reference_t<Out>&&>(*std::forward<Out>(o)) = std::forward<T>(t); // not required to be equality-preserving
};
```

2 Let E be an expression such that decltype((E)) is T, and let o be a dereferenceable object of type Out. Out and T model indirectly_writable<Out, T> only if

(2.1) If Out and T model indirectly_readable<Out> && same_as<iter_value_t<Out>, decay_t<T>>, then *o after any above assignment is equal to the value of E before the assignment.

3 After evaluating any above assignment expression, o is not required to be dereferenceable.

4 If E is an xvalue (7.2.1), the resulting state of the object it denotes is valid but unspecified (16.4.6.16).

5 [Note 1: The only valid use of an operator* is on the left side of the assignment statement. Assignment through the same value of the indirectly writable type happens only once. —end note]

6 [Note 2: indirectly_writable has the awkward const_cast expressions to reject iterators with prvalue non-proxy reference types that permit rvalue assignment but do not also permit const rvalue assignment. Consequently, an iterator type I that returns std::string by value does not model indirectly_writable<I, std::string>. —end note]

23.3.4.4 Concept weakly_incrementable

The weakly_incrementable concept specifies the requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are not required to be equality-preserving, nor is the type required to be equality_comparable.

```cpp
inline constexpr bool is_integer_like = see below; // exposition only

inline constexpr bool is_signed_integer_like = see below; // exposition only
```
template<class I>
concept weakly_incrementable =
    default_initializable<I> && movable<I> &&
    requires(I i) {
        typename iter_difference_t<I>;
        requires is-signed-integer-like<iter_difference_t<I>>;
        { ++i } -> same_as<I&>;    // not required to be equality-preserving
        i++;
        // not required to be equality-preserving
    };

2 A type I is an integer-class type if it is in a set of implementation-defined class types that behave as integer types do, as defined in below.

3 The range of representable values of an integer-class type is the continuous set of values over which it is defined. The values 0 and 1 are part of the range of every integer-class type. If any negative numbers are part of the range, the type is a signed-integer-class type; otherwise, it is an unsigned-integer-class type.

4 For every integer-class type I, let B(I) be a hypothetical extended integer type of the same signedness with the smallest width (6.8.2) capable of representing the same range of values. The width of I is equal to the width of B(I).

5 Let a and b be objects of integer-class type I, let x and y be objects of type B(I) as described above that represent the same values as a and b respectively, and let c be an lvalue of any integral type.

(5.1) — For every unary operator @ for which the expression @x is well-formed, @a shall also be well-formed and have the same value, effects, and value category as @x provided that value is representable by I. If @x has type bool, so too does @a; if @x has type B(I), then @a has type I.

(5.2) — For every assignment operator @= for which c @= x is well-formed, c @= a shall also be well-formed and shall have the same value and effects as c @= x. The expression c @= a shall be an lvalue referring to c.

(5.3) — For every binary operator @ for which x @ y is well-formed, a @ b shall also be well-formed and shall have the same value, effects, and value category as x @ y provided that value is representable by I. If x @ y has type bool, so too does a @ b; if x @ y has type B(I), then a @ b has type I.

6 Expressions of integer-class type are explicitly convertible to any integral type. Expressions of integral type are both implicitly and explicitly convertible to any integer-class type. Conversions between integral and integer-class types do not exit via an exception.

7 An expression E of integer-class type I is contextually convertible to bool as if by bool(E != I(0)).

8 All integer-class types model regular (18.6) and totally_ordered (18.5.4).

9 A value-initialized object of integer-class type has value 0.

10 For every (possibly cv-qualified) integer-class type I, numeric_limits<I> is specialized such that:

(10.1) — numeric_limits<I>::is_specialized is true,

(10.2) — numeric_limits<I>::is_signed is true if and only if I is a signed-integer-class type,

(10.3) — numeric_limits<I>::is_integer is true,

(10.4) — numeric_limits<I>::is_exact is true,

(10.5) — numeric_limits<I>::digits is equal to the width of the integer-class type,

(10.6) — numeric_limits<I>::digits10 is equal to static_cast<int>(digits * log10(2)), and

(10.7) — numeric_limits<I>::min() and numeric_limits<I>::max() return the lowest and highest representable values of I, respectively, and numeric_limits<I>::lowest() returns numeric_limits<I>::min().

11 A type I is integer-like if it models integral<I> or if it is an integer-class type. A type I is signed-integer-like if it models signed_integral<I> or if it is a signed-integer-class type. A type I is unsigned-integer-like if it models unsigned_integral<I> or if it is an unsigned-integer-class type.

12 is-integer-like<I> is true if and only if I is an integer-like type. is-signed-integer-like<I> is true if and only if I is a signed-integer-like type.

13 Let i be an object of type I. When i is in the domain of both pre- and post-increment, i is said to be incrementable. I models weakly_incrementable<I> only if

§ 23.3.4.4

937
The expressions `++i` and `i++` have the same domain.

If `i` is incrementable, then both `++i` and `i++` advance `i` to the next element.

If `i` is incrementable, then `addressof(++i)` is equal to `addressof(i)`.

**Recommended practice:** The implementation of an algorithm on a weakly incrementable type should never attempt to pass through the same incrementable value twice; such an algorithm should be a single-pass algorithm.

[Note 1: For weakly_incrementable types, `a` equals `b` does not imply that `++a` equals `++b`. (Equality does not guarantee the substitution property or referential transparency.) Such algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. — end note]

### 23.3.4.5 Concept incrementable [iterator.concept.inc]

The `incrementable` concept specifies requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are required to be equality-preserving, and the type is required to be `equality_comparable`.

[Note 1: This supersedes the annotations on the increment expressions in the definition of weakly_incrementable. — end note]

```cpp
template<class I>
concept incrementable =
  regular<I> &&
  weakly_incrementable<I> &&
  requires(I i) {
    { i++ } -> same_as<I>; 
  };
```

Let `a` and `b` be incrementable objects of type `I`. `I` models `incrementable` only if

1. If `bool(a == b)` then `bool(a++ == b)`.
2. If `bool(a == b)` then `bool(((void)a++, a) == ++b)`.

[Note 2: The requirement that `a` equals `b` implies `++a` equals `++b` (which is not true for weakly incrementable types) allows the use of multi-pass one-directional algorithms with types that model `incrementable`. — end note]

### 23.3.4.6 Concept input_or_output_iterator [iterator.concept.iterator]

The `input_or_output_iterator` concept forms the basis of the iterator concept taxonomy; every iterator models `input_or_output_iterator`. This concept specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to compare iterators with sentinels (23.3.4.7), to read (23.3.4.9) or write (23.3.4.10), or to provide a richer set of iterator movements (23.3.4.11, 23.3.4.12, 23.3.4.13).

```cpp
template<class I>
concept input_or_output_iterator =
  requires(I i) {
    { *i } -> can-reference;
  } &&
  weakly_incrementable<I>;
```

[Note 1: Unlike the Cpp17Iterator requirements, the `input_or_output_iterator` concept does not require copyability. — end note]

### 23.3.4.7 Concept sentinel_for [iterator.concept.sentinel]

The `sentinel_for` concept specifies the relationship between an `input_or_output_iterator` type and a `semiregular` type whose values denote a range.

```cpp
template<class S, class I>
concept sentinel_for =
  semiregular<S> &&
  input_or_output_iterator<I> &&
  weakly-equality-comparable-with<S, I>; // see 18.5.3
```

Let `s` and `i` be values of type `S` and `I` such that `[i, s)` denotes a range. Types `S` and `I` model `sentinel_for<S, I>` only if

1. `i == s` is well-defined.
— If bool(i != s) then i is dereferenceable and [++i, s) denotes a range.

The domain of == is not static. Given an iterator i and sentinel s such that [i, s) denotes a range and i != s, i and s are not required to continue to denote a range after incrementing any other iterator equal to i. Consequently, i == s is no longer required to be well-defined.

23.3.4.8 Concept sized_sentinel_for [iterator.concept.sizedsentinel]

The sized_sentinel_for concept specifies requirements on an input_or_output_iterator type I and a corresponding sentinel_for<I> that allow the use of the − operator to compute the distance between them in constant time.

```cpp
template<class S, class I>
concept sized_sentinel_for =
    sentinel_for<S, I> &&
    !disable_sized_sentinel_for<remove_cv_t<S>, remove_cv_t<I>> &&
    requires(const I& i, const S& s) {
        { s - i } -> same_as<iter_difference_t<I>>;
        { i - s } -> same_as<iter_difference_t<I>>;
    };
```

Let i be an iterator of type I, and s a sentinel of type S such that [i, s) denotes a range. Let N be the smallest number of applications of ++i necessary to make bool(i == s) be true. S and I model sized_sentinel_for only if

1. If N is representable by iter_difference_t<I>, then s - i is well-defined and equals N.
2. If −N is representable by iter_difference_t<I>, then i - s is well-defined and equals −N.

```
template<class S, class I>
inline constexpr bool disable_sized_sentinel_for = false;
```

Remarks: Pursuant to 16.4.5.2.1, users may specialize disable_sized_sentinel_for for cv-unqualified non-array object types S and I if S and/or I is a program-defined type. Such specializations shall be usable in constant expressions (7.7) and have type const bool.

[Note 1: disable_sized_sentinel_for allows use of sentinels and iterators with the library that satisfy but do not in fact model sized_sentinel_for. —end note]

[Example 1: The sized_sentinel_for concept is modeled by pairs of random_access_iterators (23.3.4.13) and by counted iterators and their sentinels (23.5.6.1). —end example]

23.3.4.9 Concept input_iterator [iterator.concept.input]

The input_iterator concept defines requirements for a type whose referenced values can be read (from the requirement for indirectly_readable (23.3.4.2)) and which can be both pre- and post-incremented.

```
template<class I>
concept input_iterator =
    input_or_output_iterator<I> &&
    indirectly_readable<I> &&
    requires { typename ITER_CONCEPT(I); } &&
    derived_from<ITER_CONCEPT(I), input_iterator_tag>;
```

Note 1: Unlike the Cpp17InputIterator requirements (23.3.5.3), the input_iterator concept does not need equality comparison since iterators are typically compared to sentinels. —end note]

23.3.4.10 Concept output_iterator [iterator.concept.output]

The output_iterator concept defines requirements for a type that can be used to write values (from the requirement for indirectly_writable (23.3.4.3)) and which can be both pre- and post-incremented.

```
template<class I, class T>
concept output_iterator =
    input_or_output_iterator<I> &&
    indirectly_writable<I, T> &&
    requires(I i, T&& t) {
        *i++ = std::forward<T>(t); // not required to be equality-preserving
    };
```

[Note 1: Output iterators are not required to model equality_comparable. — end note]
Let \( E \) be an expression such that \( \text{decltype}(E) \) is \( T \), and let \( i \) be a dereferenceable object of type \( I \). \( I \) and \( T \) model output_iterator\(<I, T> \) only if \( *i++ = E \); has effects equivalent to:

\[
\begin{align*}
* &i = E; \\
++ &i;
\end{align*}
\]

Recommended practice: The implementation of an algorithm on output iterators should never attempt to pass through the same iterator twice; such an algorithm should be a single-pass algorithm.

23.3.4.11 Concept forward_iterator

The forward_iterator concept adds copyability, equality comparison, and the multi-pass guarantee, specified below.

\[
\text{template<class } I > \\
\text{concept forward_iterator =} \\
\text{input_iterator}\(<I> \text{&}\&} \\
\text{derived_from<ITER_CONCEPT(I), forward_iterator_tag}\text{&}\&} \\
\text{incrementable}\(<I> \text{&}\&} \\
\text{sentinel_for}\(<I>, I>); \\
\]

The domain of \( == \) for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators of the same type may be compared and shall compare equal to other value-initialized iterators of the same type.

[Note 1: Value-initialized iterators behave as if they refer past the end of the same empty sequence. — end note]

Pointers and references obtained from a forward iterator into a range \([i, s)\] shall remain valid while \([i, s)\] continues to denote a range.

Two dereferenceable iterators \( a \) and \( b \) of type \( X \) offer the multi-pass guarantee if:

\[
\begin{align*}
&\text{(4.1)} \quad \text{-- } a \text{ } == \text{ } b \text{ } \Rightarrow \text{ } *++a \text{ } == \text{ } *++b \text{ } \text{and} \\
&\text{(4.2)} \quad \text{the expression } (\text{void}[])\{x \}\{(++x;)(a), *a\}\text{ is equivalent to the expression } *a. \\
\]

[Note 2: The requirement that \( a \text{ } == \text{ } b \text{ } \Rightarrow \text{ } *++a \text{ } == \text{ } *++b \text{ } \text{and} \text{ the removal of the restrictions on the number of assignments through a mutable iterator (which applies to output iterators) allow the use of multi-pass one-directional algorithms with forward iterators. — end note}]

23.3.4.12 Concept bidirectional_iterator

The bidirectional_iterator concept adds the ability to move an iterator backward as well as forward.

\[
\text{template<class } I > \\
\text{concept bidirectional_iterator =} \\
\text{forward_iterator}\(<I> \text{&}\&} \\
\text{derived_from<ITER_CONCEPT(I), bidirectional_iterator_tag}\text{&}\&} \\
\text{requires(I i) } \{ \{ --i\} \text{ } \rightarrow \text{ } \text{same_as}\(<I>\}; \\
\text{\{ i--\} } \rightarrow \text{ } \text{same_as}\(<I>\}; \\
\}; \\
\]

A bidirectional iterator \( r \) is decrementable if and only if there exists some \( q \) such that \( ++q \text{ } == \text{ } r \). Decrementable iterators \( r \) shall be in the domain of the expressions \( --r \text{ and } r-- \).

Let \( a \) and \( b \) be equal objects of type \( I \). \( I \) models bidirectional_iterator only if:

\[
\begin{align*}
&\text{(3.1)} \quad \text{If } a \text{ and } b \text{ are decrementable, then all of the following are true:} \\
&\text{(3.1.1)} \quad \text{addressof}(-a) \text{ } == \text{ } \text{addressof} \text{ } (a) \\
&\text{(3.1.2)} \quad \text{bool}(a-- \text{ } == \text{ } b) \\
&\text{(3.1.3)} \quad \text{after evaluating both } a-- \text{ and } --b, \text{ } \text{bool} \text{ } (a \text{ } == \text{ } b) \text{ is still true} \\
&\text{(3.1.4)} \quad \text{bool}(++(--a) \text{ } == \text{ } b) \\
&\text{(3.2)} \quad \text{If } a \text{ and } b \text{ are decrementable, then } \text{bool} \text{ } ((--a) \text{ } == \text{ } b). \\
\]

23.3.4.13 Concept random_access_iterator

The random_access_iterator concept adds support for constant-time advancement with \( +, +\text{, } --\), and \( \text{-} \), as well as the computation of distance in constant time with \( \text{-} \). Random access iterators also support array notation via subscripting.
template<class I>
concept random_access_iterator =
   bidirectional_iterator<I> &&
   derived_from<ITER_CONCEPT(I), random_access_iterator_tag> &&
   totally_ordered<I> &&
sized_sentinel_for<I, I> &&
requires(I i, const I j, const iter_difference_t<I> n) {
   { i += n } -> same_as<I&>;
   { j + n } -> same_as<I>;
   { n + j } -> same_as<I>;
   { i -= n } -> same_as<I&>;
   { j - n } -> same_as<I>;
   { j[n] } -> same_as<iter_reference_t<I>>;
};

Let a and b be valid iterators of type I such that b is reachable from a after n applications of ++a, let D be
iter_difference_t<I>, and let n denote a value of type D. I models random_access_iterator only if

(2.1) — (a += n) is equal to b.
(2.2) — addressof(a += n) is equal to addressof(a).
(2.3) — (a + n) is equal to (a += n).
(2.4) — For any two positive values x and y of type D, if (a + D(x + y)) is valid, then (a + D(x + y)) is
equal to ((a + x) + y).
(2.5) — (a + D(0)) is equal to a.
(2.6) — If (a + D(n - 1)) is valid, then (a + n) is equal to [](I c){ return ++c; }(a + D(n - 1)).
(2.7) — (b += D(-n)) is equal to a.
(2.8) — (b -= n) is equal to a.
(2.9) — addressof(b -= n) is equal to addressof(b).
(2.10) — (b - n) is equal to (b -= n).
(2.11) — If b is dereferenceable, then a[n] is valid and is equal to *b.
(2.12) — bool(a <= b) is true.

23.3.4.14 Concept contiguous_iterator [iterator.concept.contiguous]

The contiguous_iterator concept provides a guarantee that the denoted elements are stored contiguously
in memory.

template<class I>
concept contiguous_iterator =
   random_access_iterator<I> &&
   derived_from<ITER_CONCEPT(I), contiguous_iterator_tag> &&
   is_lvalue_reference_v<iter_reference_t<I> > &&
   same_as<iter_value_t<I>, remove_cvref_t<iter_reference_t<I>> > &&
requires(const I& i) {
   { to_address(i) } -> same_as<add_pointer_t<iter_reference_t<I>> >;
};

Let a and b be dereferenceable iterators and c be a non-dereferenceable iterator of type I such that b
is reachable from a and c is reachable from b, and let D be iter_difference_t<I>. The type I models
contiguous_iterator only if

(2.1) — to_address(a) == addressof(*a),
(2.2) — to_address(b) == to_address(a) + D(b - a), and
(2.3) — to_address(c) == to_address(a) + D(c - a).

23.3.5 C++17 iterator requirements [iterator.cpp17]

23.3.5.1 General [iterator.cpp17.general]

In the following sections, a and b denote values of type X or const X, difference_type and reference refer
to the types iterator_traits<X>::difference_type and iterator_traits<X>::reference, respectively,
23.3.5.2 Cpp17Iterator

The Cpp17Iterator requirements form the basis of the iterator taxonomy; every iterator meets the Cpp17Iterator requirements. This set of requirements specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to read (23.3.5.3) or write (23.3.5.4) values, or to provide a richer set of iterator movements (23.3.5.5, 23.3.5.6, 23.3.5.7).

A type X meets the Cpp17Iterator requirements if:

1. X meets the Cpp17CopyConstructible, Cpp17CopyAssignable, and Cpp17Destructible requirements (16.4.4.2) and lvalues of type X are swappable (16.4.4.3), and
2. iterator_traits<X>::difference_type is a signed integer type or void, and
3. the expressions in Table 84 are valid and have the indicated semantics.

Table 84: Cpp17Iterator requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>*r</td>
<td>unspecified</td>
<td></td>
<td>Preconditions: r is dereferenceable.</td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

23.3.5.3 Input iterators

A class or pointer type X meets the requirements of an input iterator for the value type T if X meets the Cpp17Iterator (23.3.5.2) and Cpp17EqualityComparable (Table 25) requirements and the expressions in Table 85 are valid and have the indicated semantics.

In Table 85, the term the domain of == is used in the ordinary mathematical sense to denote the set of values over which == is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of == for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of == and !=.

[Example 1: The call find(a,b,x) is defined only if the value of a has the property p defined as follows: b has property p and a value i has property p if (*i==x) or if (*i!=x and ++i has property p). — end example]

Table 85: Cpp17InputIterator requirements (in addition to Cpp17Iterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a != b</td>
<td>contextually convertible to bool</td>
<td>![a == b]</td>
<td>Preconditions: (a, b) is in the domain of ==.</td>
</tr>
<tr>
<td>*a</td>
<td>reference, convertible to T</td>
<td></td>
<td>Preconditions: a is dereferenceable. The expression (void)*a, *a is equivalent to *a. If a == b and (a, b) is in the domain of == then *a is equivalent to *b.</td>
</tr>
<tr>
<td>a-&gt;m</td>
<td>(*a).m</td>
<td></td>
<td>Preconditions: a is dereferenceable.</td>
</tr>
</tbody>
</table>
Table 85: Cpp17InputIterator requirements (in addition to Cpp17Iterator) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td></td>
<td>Preconditions: r is dereferenceable. Postconditions: r is dereferenceable or r is past-the-end; any copies of the previous value of r are no longer required to be dereferenceable nor to be in the domain of ==.</td>
</tr>
<tr>
<td>(void)r++</td>
<td></td>
<td></td>
<td>equivalent to (void)++r</td>
</tr>
<tr>
<td>*r++</td>
<td>convertible to T</td>
<td>{ T tmp = *r; ++r; return tmp; }</td>
<td>Remarks: After this operation r is not required to be dereferenceable. Postconditions: r is incrementable.</td>
</tr>
</tbody>
</table>

3 Recommended practice: The implementation of an algorithm on input iterators should never attempt to pass through the same iterator twice; such an algorithm should be a single pass algorithm.

[Note 1: For input iterators, a == b does not imply ++a == ++b. (Equality does not guarantee the substitution property or referential transparency.) Value type T is not required to be a Cpp17CopyAssignable type (Table 31). Such an algorithm can be used with istreams as the source of the input data through the istream_iterator class template. — end note]

23.3.5.4 Output iterators [output.iterators]

1 A class or pointer type X meets the requirements of an output iterator if X meets the Cpp17Iterator requirements (23.3.5.2) and the expressions in Table 86 are valid and have the indicated semantics.

Table 86: Cpp17OutputIterator requirements (in addition to Cpp17Iterator) [tab:outputiterator]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>*r = o</td>
<td>result is not used</td>
<td></td>
<td>Remarks: After this operation r is not required to be dereferenceable. Postconditions: r is incrementable.</td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td>addressof(r) == addressof(++r). Remarks: After this operation r is not required to be dereferenceable. Postconditions: r is incrementable.</td>
<td></td>
</tr>
<tr>
<td>r++</td>
<td>convertible to const X&amp;</td>
<td>{ X tmp = r; ++r; return tmp; }</td>
<td>Remarks: After this operation r is not required to be dereferenceable. Postconditions: r is incrementable.</td>
</tr>
<tr>
<td>*r++ = o</td>
<td>result is not used</td>
<td></td>
<td>Remarks: After this operation r is not required to be dereferenceable. Postconditions: r is incrementable.</td>
</tr>
</tbody>
</table>
Recommended practice: The implementation of an algorithm on output iterators should never attempt to pass through the same iterator twice; such an algorithm should be a single-pass algorithm.

[Note 1: The only valid use of an operator* is on the left side of the assignment statement. Assignment through the same value of the iterator happens only once. Equality and inequality might not be defined. — end note]

23.3.5.5 Forward iterators

A class or pointer type \( X \) meets the requirements of a forward iterator if

(1.1) — \( X \) meets the Cpp17InputIterator requirements (23.3.5.3),
(1.2) — \( X \) meets the Cpp17DefaultConstructible requirements (16.4.4.2),
(1.3) — if \( X \) is a mutable iterator, reference is a reference to \( T \); if \( X \) is a constant iterator, reference is a reference to const \( T \),
(1.4) — the expressions in Table 87 are valid and have the indicated semantics, and
(1.5) — objects of type \( X \) offer the multi-pass guarantee, described below.

The domain of == for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators may be compared and shall compare equal to other value-initialized iterators of the same type.

[Note 1: Value-initialized iterators behave as if they refer past the end of the same empty sequence. — end note]

Two dereferenceable iterators \( a \) and \( b \) of type \( X \) offer the multi-pass guarantee if:

(3.1) — \( a == b \) implies \( ++a == ++b \) and
(3.2) — \( X \) is a pointer type or the expression \((\text{void})++X(a), *a\) is equivalent to the expression \(*a\).

[Note 2: The requirement that \( a == b \) implies \( ++a == ++b \) (which is not true for input and output iterators) and the removal of the restrictions on the number of the assignments through a mutable iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators. — end note]

Table 87: Cpp17ForwardIterator requirements (in addition to Cpp17InputIterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r++ )</td>
<td>convertible to ( X&amp; )</td>
<td>( { X \ tmp = r; ) const ( X&amp; ) ( ++r; ) return ( \ tmp; } )</td>
<td></td>
</tr>
<tr>
<td>( *r++ )</td>
<td>reference</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. If \( a \) and \( b \) are equal, then either \( a \) and \( b \) are both dereferenceable or else neither is dereferenceable.
6. If \( a \) and \( b \) are both dereferenceable, then \( a == b \) if and only if \( *a \) and \( *b \) are bound to the same object.

23.3.5.6 Bidirectional iterators

A class or pointer type \( X \) meets the requirements of a bidirectional iterator if, in addition to meeting the Cpp17ForwardIterator requirements, the following expressions are valid as shown in Table 88.

[Note 1: Bidirectional iterators allow algorithms to move iterators backward as well as forward. — end note]

23.3.5.7 Random access iterators

A class or pointer type \( X \) meets the requirements of a random access iterator if, in addition to meeting the Cpp17BidirectionalIterator requirements, the following expressions are valid as shown in Table 89.

§ 23.3.5.7
### Table 88: Cpp17BidirectionalIterator requirements (in addition to Cpp17ForwardIterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--r</code></td>
<td><code>X&amp;</code></td>
<td>Preconditions: there exists <code>s</code> such that <code>r == ++s</code>. Postconditions: <code>r</code> is dereferenceable.</td>
<td></td>
</tr>
<tr>
<td><code>(--r)</code></td>
<td>convertible to</td>
<td><code>{ X tmp = r; --r; return tmp; }</code></td>
<td></td>
</tr>
<tr>
<td><code>*--r</code></td>
<td>reference</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 89: Cpp17RandomAccessIterator requirements (in addition to Cpp17BidirectionalIterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r += n</code></td>
<td><code>X&amp;</code></td>
<td><code>{ difference_type m = n; if (m &gt;= 0) while (m--) ++r; else while (m++) --r; return r; }</code></td>
<td></td>
</tr>
<tr>
<td><code>a + n</code></td>
<td><code>X</code></td>
<td><code>{ X tmp = a; return tmp += n; }</code></td>
<td><code>a + n == n + a</code>.</td>
</tr>
<tr>
<td><code>n + a</code></td>
<td><code>X</code></td>
<td><code>return r += -n;</code></td>
<td>Preconditions: the absolute value of <code>n</code> is in the range of representable values of <code>difference_type</code>.</td>
</tr>
<tr>
<td><code>r -= n</code></td>
<td><code>X&amp;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a - n</code></td>
<td><code>X</code></td>
<td><code>{ X tmp = a; return tmp -= n; }</code></td>
<td></td>
</tr>
<tr>
<td><code>b - a</code></td>
<td><code>difference_</code></td>
<td><code>return n</code></td>
<td>Preconditions: there exists a value <code>n</code> of type <code>difference_type</code> such that <code>a + n == b</code>. <code>b == a + (b - a)</code>.</td>
</tr>
<tr>
<td><code>a[n]</code></td>
<td>convertible to <code>*(a + n)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a &lt; b</code></td>
<td>contextually convertible to <code>bool</code></td>
<td><code>b - a &gt; 0</code></td>
<td><code>&lt;</code> is a total ordering relation</td>
</tr>
<tr>
<td><code>a &gt; b</code></td>
<td>contextually convertible to <code>bool</code></td>
<td><code>b &lt; a</code></td>
<td><code>&gt;</code> is a total ordering relation opposite to <code>&lt;</code>.</td>
</tr>
<tr>
<td><code>a &gt;= b</code></td>
<td>contextually convertible to <code>bool</code></td>
<td><code>!(a &lt; b)</code></td>
<td></td>
</tr>
</tbody>
</table>
Table 89: Cpp17RandomAccessIterator requirements (in addition to Cpp17BidirectionalIterator) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;= b</td>
<td>contextually</td>
<td>!((a &gt; b))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>convertible to bool.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

23.3.6 Indirect callable requirements

23.3.6.1 General

There are several concepts that group requirements of algorithms that take callable objects (20.14.3) as arguments.

23.3.6.2 Indirect callables

The indirect callable concepts are used to constrain those algorithms that accept callable objects (20.14.3) as arguments.

```cpp
namespace std {
    template<class F, class I>
    concept indirectly_unary_invocable =
        indirectly_readable<I> &&
        copy_constructible<F> &&
        invocable<F&, iter_value_t<I>&> &&
        invocable<F&, iter_reference_t<I>> &&
        invocable<F&, iter_common_reference_t<I>> &&
        common_reference_with<
            invoke_result_t<F&, iter_value_t<I>&>,
            invoke_result_t<F&, iter_reference_t<I>>;>

    template<class F, class I>
    concept indirectly_regular_unary_invocable =
        indirectly_readable<I> &&
        copy_constructible<F> &&
        regular_invocable<F&, iter_value_t<I>&> &&
        regular_invocable<F&, iter_reference_t<I>> &&
        regular_invocable<F&, iter_common_reference_t<I>> &&
        common_reference_with<
            invoke_result_t<F&, iter_value_t<I>&>,
            invoke_result_t<F&, iter_reference_t<I>>;>

    template<class F, class I1, class I2>
    concept indirect_binary_predicate =
        indirectly_readable<I1> && indirectly_readable<I2> &&
        copy_constructible<F> &&
        predicate<F&, iter_value_t<I1>&, iter_value_t<I2>&> &&
        predicate<F&, iter_value_t<I1>&, iter_reference_t<I2>> &&
        predicate<F&, iter_reference_t<I1>, iter_value_t<I2>&> &&
        predicate<F&, iter_reference_t<I1>, iter_reference_t<I2>> &&
        predicate<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;>

    template<class F, class I1, class I2 = I1>
    concept indirect_equivalence_relation =
        indirectly_readable<I1> && indirectly_readable<I2> &&
        copy_constructible<F> &&
        equivalence_relation<F&, iter_value_t<I1>&, iter_value_t<I2>&> &&
```
equivalence_relation<F&, iter_value_t<I1>&, iter_reference_t<I2>>& &&
equivalence_relation<F&, iter_reference_t<I1>, iter_value_t<I2>&> &&
equivalence_relation<F&, iter_reference_t<I1>, iter_reference_t<I2>> &&
equivalence_relation<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

template<class F, class I1, class I2 = I1>
concept indirectly_strict_weak_order =
  indirectly_readable<I1> && indirectly_readable<I2> &&
copy_constructible<F> &&
strict_weak_order<F&, iter_value_t<I1>&, iter_value_t<I2>&> &&
strict_weak_order<F&, iter_value_t<I1>&, iter_reference_t<I2>> &&
strict_weak_order<F&, iter_reference_t<I1>, iter_value_t<I2>&> &&
strict_weak_order<F&, iter_reference_t<I1>, iter_reference_t<I2>> &&
strict_weak_order<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

23.3.6.3 Class template projected

Class template projected is used to constrain algorithms that accept callable objects and projections (3.44). It combines a indirectly_readable type I and a callable object type Proj into a new indirectly_readable type whose reference type is the result of applying Proj to the iter_reference_t of I.

namespace std {
  template<indirectly_readable I, indirectly_regular_unary_invocable<I> Proj>
  struct projected {
    using value_type = remove_cvref_t<indirect_result_t<Proj&, I>>;
    indirect_result_t<Proj&, I> operator*() const; // not defined
  };
}

23.3.7 Common algorithm requirements [alg.req]

23.3.7.1 General [alg.req.general]

There are several additional iterator concepts that are commonly applied to families of algorithms. These group together iterator requirements of algorithm families. There are three relational concepts that specify how element values are transferred between indirectly_readable and indirectly_writable types: indirectly_movable, indirectly_copyable, and indirectly_swappable. There are three relational concepts for rearrangements: permutable, mergeable, and sortable. There is one relational concept for comparing values from different sequences: indirectly_comparable.

[Note 1: The ranges::less function object type used in the concepts below imposes constraints on the concepts’ arguments in addition to those that appear in the concepts’ bodies (20.14.9). — end note]

23.3.7.2 Concept indirectly_movable [alg.req.ind.move]

The indirectly_movable concept specifies the relationship between a indirectly_readable type and a indirectly_writable type between which values may be moved.

template<class In, class Out>
concept indirectly_movable =
  indirectly_readable<In> &&
  indirectly_writable<Out, iter_rvalue_reference_t<In>>;

The indirectly_movable_storable concept augments indirectly_movable with additional requirements enabling the transfer to be performed through an intermediate object of the indirectly_readable type’s value type.

template<class In, class Out>
concept indirectly_movable_storable =
  indirectly_movable<In, Out> &&
  movable<iter_value_t<In>> &&
  constructible_from<iter_value_t<In>, iter_rvalue_reference_t<In>> &&

assignable_from<iter_value_t<In>&, iter_rvalue_reference_t<In>>;

3 Let \( i \) be a dereferenceable value of type \( \text{In} \). \( \text{In} \) and \( \text{Out} \) model `indirectly_movable_storable<\text{In}, \text{Out}>` only if after the initialization of the object \( \text{obj} \) in

\[
\text{iter_value_t<\text{In}> } \text{obj}(\text{ranges::iter_move}(\text{i}));
\]

\( \text{obj} \) is equal to the value previously denoted by \( \ast i \). If `iter_rvalue_reference_t<\text{In}>` is an rvalue reference type, the resulting state of the value denoted by \( \ast i \) is valid but unspecified (16.4.6.16).

23.3.7.3 Concept indirectly_copyable

The `indirectly_copyable` concept specifies the relationship between a `indirectly_readable` type and a `indirectly_writable` type between which values may be copied.

\[
\text{template<class \text{In}, class \text{Out}>}
\]

\[
\text{concept indirectly_copyable =}
\]

\[
\text{indirectly_readable<\text{In}> } \&\&
\]

\[
\text{indirectly_writable<\text{Out}, iter_reference_t<\text{In}>>>};
\]

2 The `indirectly_copyable_storable` concept augments `indirectly_copyable` with additional requirements enabling the transfer to be performed through an intermediate object of the `indirectly_readable` type’s value type. It also requires the capability to make copies of values.

\[
\text{template<class \text{In}, class \text{Out}>}
\]

\[
\text{concept indirectly_copyable_storable =}
\]

\[
\text{indirectly_copyable<\text{In}, \text{Out}> } \&\&
\]

\[
\text{indirectly_writable<\text{Out}, iter_value_t<\text{In}>>&&}
\]

\[
\text{indirectly_writable<\text{Out}, const iter_value_t<\text{In}>>&&}
\]

\[
\text{indirectly_writable<\text{Out}, iter_value_t<\text{In>>&&}
\]

\[
\text{copiable<iter_value_t<\text{In}>>&&}
\]

\[
\text{constructible_from<iter_value_t<\text{In}>}, iter_reference_t<\text{In}>>&&
\]

\[
\text{assignable_from<iter_value_t<\text{In}>}, iter_reference_t<\text{In}>>};
\]

3 Let \( i \) be a dereferenceable value of type \( \text{In} \). \( \text{In} \) and \( \text{Out} \) model `indirectly_copyable_storable<\text{In}, \text{Out}>` only if after the initialization of the object \( \text{obj} \) in

\[
\text{iter_value_t<\text{In}> } \text{obj}(\ast i);
\]

\( \text{obj} \) is equal to the value previously denoted by \( \ast i \). If `iter_reference_t<\text{In}>` is an rvalue reference type, the resulting state of the value denoted by \( \ast i \) is valid but unspecified (16.4.6.16).

23.3.7.4 Concept indirectly_swappable

The `indirectly_swappable` concept specifies a swappable relationship between the values referenced by two `indirectly_readable` types.

\[
\text{template<class \text{I1}, class \text{I2} = \text{I1}>}
\]

\[
\text{concept indirectly_swappable =}
\]

\[
\text{indirectly_readable<\text{I1}> } \&\&
\]

\[
\text{indirectly_readable<\text{I2}> } \&\&
\]

\[
\text{requires(const \text{I1} \text{i1}, const \text{I2} \text{i2}) {}
\]

\[
\text{ranges::iter_swap(\text{i1}, \text{i1});}
\]

\[
\text{ranges::iter_swap(\text{i2}, \text{i2});}
\]

\[
\text{ranges::iter_swap(\text{i1}, \text{i2});}
\]

\[
\text{ranges::iter_swap(\text{i2}, \text{i1});}
\}
\]

23.3.7.5 Concept indirectly_comparable

The `indirectly_comparable` concept specifies the common requirements of algorithms that compare values from two different sequences.

\[
\text{template<class \text{I1}, class \text{I2}, class \text{R}, class \text{P1} = \text{identity},}
\]

\[
\text{class \text{P2} = \text{identity}>}
\]

\[
\text{concept indirectly_comparable =}
\]

\[
\text{indirect_binary_predicate<\text{R}, projected<\text{I1}, \text{P1}>, projected<\text{I2}, \text{P2}>>};
\]

23.3.7.6 Concept permutable

The `permutable` concept specifies the common requirements of algorithms that reorder elements in place by moving or swapping them.
template<class I>
concept permutable =
    forward_iterator<I> &&
    indirectly_movable_storable<I, I> &&
    indirectly_swappable<I, I>;

23.3.7.7 Concept mergeable [alg.req.mergeable]
1 The mergeable concept specifies the requirements of algorithms that merge sorted sequences into an output sequence by copying elements.

    template<class I1, class I2, class Out, class R = ranges::less,
             class P1 = identity, class P2 = identity>
    concept mergeable =
        input_iterator<I1> &&
        input_iterator<I2> &&
        weakly_incrementable<Out> &&
        indirectly_copyable<I1, Out> &&
        indirectly_copyable<I2, Out> &&
        indirect_strict_weak_order<R, projected<I1, P1>, projected<I2, P2>>;  

23.3.7.8 Concept sortable [alg.req.sortable]
1 The sortable concept specifies the common requirements of algorithms that permute sequences into ordered sequences (e.g., sort).

    template<class I, class R = ranges::less, class P = identity>
    concept sortable =
        permutable<I> &&
        indirect_strict_weak_order<R, projected<I, P>>;  

23.4 Iterator primitives [iterator.primitives]

23.4.1 General [iterator.primitives.general]
1 To simplify the use of iterators, the library provides several classes and functions.

23.4.2 Standard iterator tags [std.iterator.tags]
1 It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces category tag classes which are used as compile time tags for algorithm selection. They are: output_iterator_tag, input_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag, random_access_iterator_tag, and contiguous_iterator_tag. For every iterator of type I, iterator_traits<I>::iterator_category shall be defined to be a category tag that describes the iterator’s behavior. Additionally, iterator_traits<I>::iterator_concept may be used to indicate conformance to the iterator concepts (23.3.4).

    namespace std {
        struct output_iterator_tag { };  
        struct input_iterator_tag { };  
        struct forward_iterator_tag: public input_iterator_tag { };  
        struct bidirectional_iterator_tag: public forward_iterator_tag { };  
        struct random_access_iterator_tag: public bidirectional_iterator_tag { };  
        struct contiguous_iterator_tag: public random_access_iterator_tag { };  
    }

2 [Example 1] For a program-defined iterator BinaryTreeIterator, it could be included into the bidirectional iterator category by specializing the iterator_traits template:

    template<class T> struct iterator_traits<BinaryTreeIterator<T>> {
        using iterator_category = bidirectional_iterator_tag;
        using difference_type = ptrdiff_t;
        using value_type = T;
        using pointer = T*;
        using reference = T&;
    };
Example 2: If `evolve()` is well-defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is as follows:

```cpp
template<class BidirectionalIterator>
inline void
evolve(BidirectionalIterator first, BidirectionalIterator last) {
    evolve(first, last,
        typename iterator_traits<BidirectionalIterator>::iterator_category());
}
```

```cpp
template<class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last,
    bidirectional_iterator_tag) {
    // more generic, but less efficient algorithm
}
```

```cpp
template<class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last,
    random_access_iterator_tag) {
    // more efficient, but less generic algorithm
}
```

---

23.4.3 Iterator operations

Since only random access iterators provide + and – operators, the library provides two function templates `advance` and `distance`. These function templates use + and – for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations.

```cpp
template<class InputIterator, class Distance>
constexpr void advance(InputIterator& i, Distance n);
```

**Preconditions:** 
n is negative only for bidirectional iterators.

**Effects:** Increments i by n if n is non-negative, and decrements i by –n otherwise.

```cpp
template<class InputIterator>
constexpr typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

**Preconditions:** last is reachable from first, or InputIterator meets the Cpp17RandomAccessIterator requirements and first is reachable from last.

**Effects:** If InputIterator meets the Cpp17RandomAccessIterator requirements, returns (last – first); otherwise, returns the number of increments needed to get from first to last.

```cpp
template<class InputIterator>
constexpr InputIterator next(InputIterator x,
    typename iterator_traits<InputIterator>::difference_type n = 1);
```

**Effects:** Equivalent to: `advance(x, n); return x;`

```cpp
template<class BidirectionalIterator>
constexpr BidirectionalIterator prev(BidirectionalIterator x,
    typename iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

**Effects:** Equivalent to: `advance(x, –n); return x;`

23.4.4 Range iterator operations

The library includes the function templates `ranges::advance`, `ranges::distance`, `ranges::next`, and `ranges::prev` to manipulate iterators. These operations adapt to the set of operators provided by each iterator category to provide the most efficient implementation possible for a concrete iterator type.

§ 23.4.4.1
Example 1: ranges::advance uses the + operator to move a random_access_iterator forward n steps in constant time. For an iterator type that does not model random_access_iterator, ranges::advance instead performs n individual increments with the ++ operator. — end example

The function templates defined in 23.4.4 are not found by argument-dependent name lookup (6.5.3). When found by unqualified (6.5.2) name lookup for the postfix-expression in a function call (7.6.1.3), they inhibit argument-dependent name lookup.

Example 2:
```cpp
void foo() {
    using namespace std::ranges;
    std::vector<int> vec{1,2,3};
    distance(begin(vec), end(vec)); // #1
}
```
The function call expression at #1 invokes std::ranges::distance, not std::distance, despite that (a) the iterator type returned from begin(vec) and end(vec) may be associated with namespace std and (b) std::distance is more specialized (13.7.7.3) than std::ranges::distance since the former requires its first two parameters to have the same type. — end example

The number and order of deducible template parameters for the function templates defined in 23.4.4 is unspecified, except where explicitly stated otherwise.

23.4.4.2 ranges::advance

```
template<input_or_output_iterator I>
constexpr void ranges::advance(I& i, iter_difference_t<I> n);

Preconditions: If I does not model bidirectional_iterator, n is not negative.
```

Effects:

(2.1) If I models random_access_iterator, equivalent to i += n.
(2.2) Otherwise, if n is non-negative, increments i by n.
(2.3) Otherwise, decrements i by -n.

```
template<input_or_output_iterator I, sentinel_for<I> S>
constexpr void ranges::advance(I& i, S bound);

Preconditions: [i, bound) denotes a range.
```

Effects:

(4.1) If I and S model assignable_from<I&, S>, equivalent to i = std::move(bound).
(4.2) Otherwise, if S and I model sized_sentinel_for<S, I>, equivalent to ranges::advance(i, bound - i).
(4.3) Otherwise, while bool(i != bound) is true, increments i.

```
template<input_or_output_iterator I, sentinel_for<I> S>
constexpr iter_difference_t<I> ranges::advance(I& i, iter_difference_t<I> n, S bound);

Preconditions: If n > 0, [i, bound) denotes a range. If n == 0, [i, bound) or (bound, i) denotes a range. If n < 0, [bound, i) denotes a range. I models bidirectional_iterator, and I and S model same_as<I, S>.
```

Effects:

(6.1) If S and I model sized_sentinel_for<S, I>:
(6.1.1) If |n| ≥ |bound - i|, equivalent to ranges::advance(i, bound).
(6.1.2) Otherwise, equivalent to ranges::advance(i, n).
(6.2) Otherwise,
(6.2.1) if n is non-negative, while bool(i != bound) is true, increments i but at most n times.
(6.2.2) Otherwise, while bool(i != bound) is true, decrements i but at most -n times.

Returns: n - M, where M is the difference between the ending and starting positions of i.

§ 23.4.4.2
23.4.4.3 `ranges::distance`[range.iter.op.distance]

template<input_or_output_iterator I, sentinel_for<I> S>
constexpr iter_difference_t<I> ranges::distance(I first, S last);

1 Preconditions: `[first, last)` denotes a range, or `[last, first)` denotes a range and S and I model same_as<S, I> && sized_sentinel_for<S, I>.

2 Effects: If S and I model sized_sentinel_for<S, I>, returns (last - first); otherwise, returns the number of increments needed to get from first to last.

template<range R>
constexpr range_difference_t<R> ranges::distance(R&& r);

3 Effects: If R models sized_range, equivalent to:

   return static_cast<range_difference_t<R>>(ranges::size(r)); // 24.3.10

Otherwise, equivalent to:

   return ranges::distance(ranges::begin(r), ranges::end(r)); // 24.3

23.4.4.4 `ranges::next`[range.iter.op.next]

template<input_or_output_iterator I>
constexpr I ranges::next(I x);

1 Effects: Equivalent to: `++x; return x;`

2 template<input_or_output_iterator I>
constexpr I ranges::next(I x, iter_difference_t<I> n);

2 Effects: Equivalent to: `ranges::advance(x, n); return x;`

3 template<input_or_output_iterator I, sentinel_for<I> S>
constexpr I ranges::next(I x, S bound);

3 Effects: Equivalent to: `ranges::advance(x, bound); return x;`

4 template<input_or_output_iterator I, sentinel_for<I> S>
constexpr I ranges::next(I x, iter_difference_t<I> n, S bound);

4 Effects: Equivalent to: `ranges::advance(x, n, bound); return x;`

23.4.4.5 `ranges::prev`[range.iter.op.prev]

template<bidirectional_iterator I>
constexpr I ranges::prev(I x);

1 Effects: Equivalent to: `--x; return x;`

2 template<bidirectional_iterator I>
constexpr I ranges::prev(I x, iter_difference_t<I> n);

2 Effects: Equivalent to: `ranges::advance(x, -n); return x;`

3 template<bidirectional_iterator I>
constexpr I ranges::prev(I x, iter_difference_t<I> n, I bound);

3 Effects: Equivalent to: `ranges::advance(x, -n, bound); return x;`

23.5 Iterator adaptors [predef.iterators]

23.5.1 Reverse iterators [reverse.iterators]

23.5.1.1 General [reverse.iterators.general]

1 Class template `reverse_iterator` is an iterator adaptor that iterates from the end of the sequence defined by its underlying iterator to the beginning of that sequence.
Class template reverse_iterator

namespace std {
    template<class Iterator>
    class reverse_iterator {
    public:
        using iterator_type = Iterator;
        using iterator_concept = see below;
        using iterator_category = see below;
        using value_type = iter_value_t<Iterator>;
        using difference_type = iter_difference_t<Iterator>;
        using pointer = typename iterator_traits<Iterator>::pointer;
        using reference = iter_reference_t<Iterator>;
        constexpr reverse_iterator();
        constexpr explicit reverse_iterator(Iterator x);
        template<class U> constexpr reverse_iterator(const reverse_iterator<U>& u);
        template<class U> constexpr reverse_iterator& operator=(const reverse_iterator<U>& u);
        constexpr Iterator base() const;
        constexpr reference operator*() const;
        constexpr pointer operator->() const requires see below;
        constexpr reverse_iterator& operator++();
        constexpr reverse_iterator operator++(int);
        constexpr reverse_iterator& operator--();
        constexpr reverse_iterator operator--(int);
        constexpr reverse_iterator operator+ (difference_type n) const;
        constexpr reverse_iterator& operator+=(difference_type n);
        constexpr reverse_iterator operator- (difference_type n) const;
        constexpr reverse_iterator& operator-=(difference_type n);
        constexpr unspecified operator[](difference_type n) const;
        friend constexpr iter_rvalue_reference_t<Iterator>
                iter_move(const reverse_iterator& i) noexcept(see below);
        template<indirectly_swappable<Iterator> Iterator2>
        friend constexpr void
                iter_swap(const reverse_iterator x, const reverse_iterator<Iterator2>& y) noexcept(see below);

    protected:
        Iterator current;
    };
}

1 The member typedef-name iterator_concept denotes
   — random_access_iterator_tag if Iterator models random_access_iterator, and
   — bidirectional_iterator_tag otherwise.

2 The member typedef-name iterator_category denotes
   — random_access_iterator_tag if the type iterator_traits<Iterator>::iterator_category models
class reverse_iterator [reverse.iterator]
derived_from<random_access_iterator_tag>, and
   — iterator_traits<Iterator>::iterator_category otherwise.

Requirements

1 The template parameter Iterator shall either meet the requirements of a Cpp17BidirectionalIterator (23.3.5.6)
or model bidirectional_iterator (23.3.4.12).

2 Additionally, Iterator shall either meet the requirements of a Cpp17RandomAccessIterator (23.3.5.7) or
model random_access_iterator (23.3.4.13) if the definitions of any of the members
   — operator+, operator-, operator=, operator+=, operator-= (23.5.1.7), or
   — operator[] (23.5.1.6),
or the non-member operators (23.5.1.8)

(2.3) — operator<, operator>, operator<*, operator>*, operator-, or operator+ (23.5.1.9)

are instantiated (13.9.2).

23.5.1.4 Construction and assignment

constexpr reverse_iterator();

1 Effects: Value-initializes current. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type Iterator.

constexpr explicit reverse_iterator(Iterator x);

2 Effects: Initializes current with x.

template<class U> constexpr reverse_iterator(const reverse_iterator<U>& u);

3 Effects: Initializes current with u.current.

template<class U>
constexpr reverse_iterator& operator=(const reverse_iterator<U>& u);

4 Effects: Assigns u.base() to current.

5 Returns: *this.

23.5.1.5 Conversion

constexpr Iterator base() const; // explicit

1 Returns: current.

23.5.1.6 Element access

constexpr reference operator*() const;

1 Effects: As if by:
    Iterator tmp = current;
    return *--tmp;

constexpr pointer operator->() const
requires (is_pointer_v<Iterator> || requires (const Iterator i) { i.operator->(); });

2 Effects:
(2.1) — If Iterator is a pointer type, equivalent to: return prev(current);
(2.2) — Otherwise, equivalent to: return prev(current).operator->();

constexpr unspecified operator[](difference_type n) const;

3 Returns: current[-n-1].

23.5.1.7 Navigation

constexpr reverse_iterator operator+(difference_type n) const;

1 Returns: reverse_iterator(current+n).

constexpr reverse_iterator operator-(difference_type n) const;

2 Returns: reverse_iterator(current-n).

constexpr reverse_iterator& operator++();

3 Effects: As if by: --current;

4 Returns: *this.
constexpr reverse_iterator operator++(int);

Effects: As if by:
        reverse_iterator tmp = *this;
        --current;
        return tmp;

constexpr reverse_iterator& operator--();

Effects: As if by ++current.
Returns: *this.

constexpr reverse_iterator operator--(int);

Effects: As if by:
        reverse_iterator tmp = *this;
        ++current;
        return tmp;

constexpr reverse_iterator& operator+=(difference_type n);

Effects: As if by:
        current -= n;
Returns: *this.

constexpr reverse_iterator& operator-=(difference_type n);

Effects: As if by: current += n;
Returns: *this.

23.5.1.8 Comparisons [reverse.iter.cmp]

template<class Iterator1, class Iterator2>
constexpr bool operator==(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

Constraints: x.base() == y.base() is well-formed and convertible to bool.
Returns: x.base() == y.base().

template<class Iterator1, class Iterator2>
constexpr bool operator!=(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

Constraints: x.base() != y.base() is well-formed and convertible to bool.
Returns: x.base() != y.base().

template<class Iterator1, class Iterator2>
constexpr bool operator<(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

Constraints: x.base() > y.base() is well-formed and convertible to bool.
Returns: x.base() > y.base().

template<class Iterator1, class Iterator2>
constexpr bool operator>(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

Constraints: x.base() < y.base() is well-formed and convertible to bool.
Returns: x.base() < y.base().

\[23.5.1.8\]
const reverse_iterator< Iterator2 >& y);

Constraints: x.base() >= y.base() is well-formed and convertible to bool.
Returns: x.base() >= y.base().

template<class Iterator1, class Iterator2>
constexpr bool operator==(const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);

Constraints: x.base() == y.base() is well-formed and convertible to bool.
Returns: x.base() == y.base().

template<class Iterator1, class Iterator2>
constexpr compare_three_way_result_t<Iterator1, Iterator2>
operator<=>(const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);

Returns: y.base() <=> x.base().

[Note 1: The argument order in the Returns: element is reversed because this is a reverse iterator. — end note]

23.5.1.9 Non-member functions

template<class Iterator1, class Iterator2>
constexpr auto operator-(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());

Returns: y.base() - x.base().

template<class Iterator>
constexpr reverse_iterator<Iterator> operator+(iter_difference_t<Iterator> n,
const reverse_iterator<Iterator>& x);

Returns: reverse_iterator<Iterator>(x.base() - n).

friend constexpr iter_rvalue_reference_t<Iterator> iter_move(const reverse_iterator& i) noexcept;

Effects: Equivalent to:
auto tmp = i.base();
return ranges::iter_move(--tmp);

Remarks: The expression in noexcept is equivalent to:
is_nothrow_copy_constructible_v<Iterator> &&
noexcept(ranges::iter_move(--declval<Iterator&>()))

template<indirectly_swappable<Iterator> Iterator2>
friend constexpr void iter_swap(const reverse_iterator& x,
const reverse_iterator<Iterator2>& y) noexcept;

Effects: Equivalent to:
auto xtmp = x.base();
auto ytmp = y.base();
ranges::iter_swap(--xtmp, --ytmp);

Remarks: The expression in noexcept is equivalent to:
is_nothrow_copy_constructible_v<Iterator> &&
is_nothrow_copy_constructible_v<Iterator2> &&
noexcept(ranges::iter_swap(--declval<Iterator&>(), --declval<Iterator2&>()))

template<class Iterator>
constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i);

Returns: reverse_iterator<Iterator>(i).
23.5.2 Insert iterators

23.5.2.1 General

To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called insert iterators, are provided in the library. With regular iterator classes,

```c++
while (first != last) *result++ = *first++;
```

causes a range \([first, last)\) to be copied into a range starting with \(result\). The same code with \(result\) being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the insert mode instead of the regular overwrite mode.

An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators meet the requirements of output iterators. \(\text{operator}\ast\) returns the insert iterator itself. The assignment \(\text{operator}=(\text{const } T & x)\) is defined on insert iterators to allow writing into them, it inserts \(x\) right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. \(\text{back_insert_iterator}\) inserts elements at the end of a container, \(\text{front_insert_iterator}\) inserts elements at the beginning of a container, and \(\text{insert_iterator}\) inserts elements where the iterator points to in a container. \(\text{back_inserter}, \text{front_inserter}, \text{and inserter}\) are three functions making the insert iterators out of a container.

23.5.2.2 Class template back_insert_iterator

```c++
namespace std {
  template<class Container>
  class back_insert_iterator {
    protected:
      Container* container = nullptr;

    public:
      using iterator_category = output_iterator_tag;
      using value_type = void;
      using difference_type = ptrdiff_t;
      using pointer = void;
      using reference = void;
      using container_type = Container;

      constexpr back_insert_iterator() noexcept = default;
      constexpr explicit back_insert_iterator(Container& x);
      constexpr back_insert_iterator& operator=(const typename Container::value_type& value);
      constexpr back_insert_iterator& operator=(typename Container::value_type&& value);
      constexpr back_insert_iterator& operator*();
      constexpr back_insert_iterator& operator++();
      constexpr back_insert_iterator operator++(int);
  };
}
```

23.5.2.2.1 Operations

```c++
constexpr explicit back_insert_iterator(Container& x);
```

\(\text{Effects}:\) Initializes \(\text{container}\) with \&\(x\).

```c++
constexpr back_insert_iterator& operator=(const typename Container::value_type& value);
```

\(\text{Effects}:\) As if by: \(\text{container}\rightarrow\text{push_back}(\text{value})\).

\(\text{Returns}:\) \(*\text{this}.*\)

```c++
constexpr back_insert_iterator& operator=(typename Container::value_type&& value);
```

\(\text{Effects}:\) As if by: \(\text{container}\rightarrow\text{push_back}(\text{std::move}(\text{value}))\).

\(\text{Returns}:\) \(*\text{this}.*\)

```c++
constexpr back_insert_iterator& operator*();
```

\(\text{Returns}:\) \(*\text{this}.*\)

\§ 23.5.2.2.1 957
constexpr back_insert_iterator& operator++();
constexpr back_insert_iterator operator++(int);

Returns: *this.

23.5.2.2.2 back_inserter

template<class Container>
constexpr back_insert_iterator<Container> back_inserter(Container& x);

Returns: back_insert_iterator<Container>(x).

23.5.2.3 Class template front_insert_iterator

namespace std {
    template<class Container>
    class front_insert_iterator {
        protected:
            Container* container = nullptr;

        public:
            using iterator_category = output_iterator_tag;
            using value_type = void;
            using difference_type = ptrdiff_t;
            using pointer = void;
            using reference = void;
            using container_type = Container;

            constexpr front_insert_iterator() noexcept = default;
            constexpr explicit front_insert_iterator(Container& x);
            constexpr front_insert_iterator& operator=(const typename Container::value_type& value);
            constexpr front_insert_iterator& operator=(typename Container::value_type&& value);
            constexpr front_insert_iterator& operator*();
            constexpr front_insert_iterator& operator++();
            constexpr front_insert_iterator operator++(int);
    };
}

23.5.2.3.1 Operations

constexpr explicit front_insert_iterator(Container& x);

Effects: Initializes container with addressof(x).

constexpr front_insert_iterator& operator=(const typename Container::value_type& value);

Effects: As if by: container->push_front(value);

Returns: *this.

constexpr front_insert_iterator& operator=(typename Container::value_type&& value);

Effects: As if by: container->push_front(std::move(value));

Returns: *this.

constexpr front_insert_iterator& operator*();

Returns: *this.

constexpr front_insert_iterator& operator++();
constexpr front_insert_iterator operator++(int);

Returns: *this.

23.5.2.3.2 front_inserter

template<class Container>
constexpr front_insert_iterator<Container> front_inserter(Container& x);

Returns: front_insert_iterator<Container>(x).

§ 23.5.2.3.2
23.5.2.4 Class template `insert_iterator`

```cpp
class insert_iterator {
  Container* container = nullptr;
  ranges::iterator_t<Container> iter = ranges::iterator_t<Container>();

public:
  using iterator_category = output_iterator_tag;
  using value_type = void;
  using difference_type = ptrdiff_t;
  using pointer = void;
  using reference = void;
  using container_type = Container;

  insert_iterator() = default;
  constexpr insert_iterator(Container& x, ranges::iterator_t<Container> i);
  constexpr insert_iterator& operator=(const typename Container::value_type& value);
  constexpr insert_iterator& operator=(typename Container::value_type&& value);
  constexpr insert_iterator& operator*();
  constexpr insert_iterator& operator++();
  constexpr insert_iterator& operator++(int);
};
```

23.5.2.4.1 Operations

```cpp
constexpr insert_iterator(Container& x, ranges::iterator_t<Container> i);
```

1. **Effects:** Initializes `container` with `addressof(x)` and `iter` with `i`.

```cpp
constexpr insert_iterator& operator=(const typename Container::value_type& value);
```

2. **Effects:** As if by:
   ```cpp
   iter = container->insert(iter, value);
   ++iter;
   ```

3. **Returns:** `*this`.

```cpp
constexpr insert_iterator& operator=(typename Container::value_type&& value);
```

4. **Effects:** As if by:
   ```cpp
   iter = container->insert(iter, std::move(value));
   ++iter;
   ```

5. **Returns:** `*this`.

```cpp
constexpr insert_iterator& operator*();
```

6. **Returns:** `*this`.

```cpp
constexpr insert_iterator& operator++();
```

7. **Returns:** `*this`.

23.5.2.4.2 `inserter`

```cpp
template<class Container>
constexpr insert_iterator<Container> inserter(Container& x, ranges::iterator_t<Container> i);
```

1. **Returns:** `insert_iterator<Container>(x, i)`.
23.5.3 Move iterators and sentinels

23.5.3.1 General

Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its indirection operator implicitly converts the value returned by the underlying iterator’s indirection operator to an rvalue. Some generic algorithms can be called with move iterators to replace copying with moving.

[Example 1:

```cpp
list<string> s;
// populate the list s
vector<string> v1(s.begin(), s.end());  // copies strings into v1
vector<string> v2(make_move_iterator(s.begin()),
    make_move_iterator(s.end()));  // moves strings into v2
```
—end example]

23.5.3.2 Class template `move_iterator`

```cpp
namespace std {
    template<class Iterator>
    class move_iterator {
        public:
            using iterator_type = Iterator;
            using iterator_concept = input_iterator_tag;
            using iterator_category = see below;
            using value_type = iter_value_t<Iterator>;
            using difference_type = iter_difference_t<Iterator>;
            using pointer = Iterator;
            using reference = iter_rvalue_reference_t<Iterator>;

            constexpr move_iterator();
            constexpr explicit move_iterator(Iterator i);
            template<class U> constexpr move_iterator(const move_iterator<U>& u);
            template<class U> constexpr move_iterator& operator=(const move_iterator<U>& u);

            constexpr iterator_type base() const &;
            constexpr iterator_type base() &&;
            constexpr reference operator*() const;
            constexpr move_iterator& operator++();
            constexpr auto operator++(int);
            constexpr move_iterator& operator--();
            constexpr move_iterator operator--(int);
            constexpr move_iterator operator+(difference_type n) const;
            constexpr move_iterator& operator+=(difference_type n);
            constexpr move_iterator operator-(difference_type n) const;
            constexpr move_iterator& operator-=(difference_type n);
            constexpr reference operator[](difference_type n) const;

            template<sentinel_for<Iterator> S>
            friend constexpr bool operator==(const move_iterator& x, const move_sentinel<S>& y);
            template<sized_sentinel_for<Iterator> S>
            friend constexpr iter_difference_t<Iterator> operator-(const move_sentinel<S>& x, const move_iterator& y);
            template<sized_sentinel_for<Iterator> S>
            friend constexpr iter_difference_t<Iterator> operator-(const move_iterator& x, const move_sentinel<S>& y);
            friend constexpr iter_rvalue_reference_t<Iterator> iter_move(const move_iterator& i)
                noexcept(noexcept(ranges::iter_move(i.current)));
```
template<indirectly_swappable<Iterator> Iterator2>  
friend constexpr void  
    iter_swap(const move_iterator& x, const move_iterator<Iterator2>& y)  
    noexcept(noexcept(ranges::iter_swap(x.current, y.current)));

private:  
    Iterator current;  // exposition only
};

1 The member typedef-name iterator_category denotes

(1.1) — random_access_iterator_tag if the type iterator_traits<Iterator>::iterator_category models derived_from<random_access_iterator_tag>, and

(1.2) — iterator_traits<Iterator>::iterator_category otherwise.

23.5.3.3 Requirements

The template parameter Iterator shall either meet the Cpp17InputIterator requirements (23.3.5.3) or model input_iterator (23.3.4.9). Additionally, if any of the bidirectional traversal functions are instantiated, the template parameter shall either meet the Cpp17BidirectionalIterator requirements (23.3.5.6) or model bidirectional_iterator (23.3.4.12). If any of the random access traversal functions are instantiated, the template parameter shall either meet the Cpp17RandomAccessIterator requirements (23.3.5.7) or model random_access_iterator (23.3.4.13).

23.5.3.4 Construction and assignment

constexpr move_iterator();

1 Effects: Constructs a move_iterator, value-initializing current. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type Iterator.

constexpr explicit move_iterator(Iterator i);

2 Effects: Constructs a move_iterator, initializing current with std::move(i).

template<class U> constexpr move_iterator(const move_iterator<U>& u);

3 Mandates: U is convertible to Iterator.

4 Effects: Constructs a move_iterator, initializing current with u.base().

template<class U> constexpr move_iterator& operator=(const move_iterator<U>& u);

5 Mandates: U is convertible to Iterator.

6 Effects: Assigns u.base() to current.

23.5.3.5 Conversion

constexpr Iterator base() const &;

1 Constraints: Iterator satisfies copy_constructible.

2 Preconditions: Iterator models copy_constructible.

3 Returns: current.

constexpr Iterator base() &&;

4 Returns: std::move(current).

23.5.3.6 Element access

constexpr reference operator*() const;

1 Effects: Equivalent to: return ranges::iter_move(current);

constexpr reference operator[](difference_type n) const;

2 Effects: Equivalent to: return ranges::iter_move(current + n);
23.5.3.7 Navigation

```cpp
constexpr move_iterator& operator++();
```

**Effects:** As if by `++current`.

**Returns:** `*this`.

```cpp
constexpr auto operator++(int);
```

**Effects:** If `Iterator` models `forward_iterator`, equivalent to:

```cpp
move_iterator tmp = *this;
++current;
return tmp;
```

Otherwise, equivalent to `++current`.

```cpp
constexpr move_iterator& operator--();
```

**Effects:** As if by `--current`.

**Returns:** `*this`.

```cpp
constexpr move_iterator operator--(int);
```

**Effects:** As if by:

```cpp
move_iterator tmp = *this;
--current;
return tmp;
```

```cpp
constexpr move_iterator operator+(difference_type n) const;
```  
**Returns:** `move_iterator(current + n)`.

```cpp
constexpr move_iterator& operator+=(difference_type n);
```  
**Effects:** As if by:

```cpp
current += n;
```

**Returns:** `*this`.

```cpp
constexpr move_iterator operator-(difference_type n) const;
```  
**Returns:** `move_iterator(current - n)`.

```cpp
constexpr move_iterator& operator-=(difference_type n);
```  
**Effects:** As if by:

```cpp
current -= n;
```

**Returns:** `*this`.

23.5.3.8 Comparisons

```cpp
template<class Iterator1, class Iterator2>
constexpr bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```  
**Constraints:** `x.base() == y.base()` is well-formed and convertible to `bool`.

**Returns:** `x.base() == y.base()`.

```cpp
template<class Iterator1, class Iterator2>
constexpr bool operator< (const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```  
**Constraints:** `x.base() < y.base()` is well-formed and convertible to `bool`.

**Returns:** `x.base() < y.base()`.

```cpp
template<class Iterator1, class Iterator2>
constexpr bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```  
**Constraints:** `y.base() < x.base()` is well-formed and convertible to `bool`.

**Returns:** `y < x`.

§ 23.5.3.8
template<class Iterator1, class Iterator2>
constexpr bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Constraints:
y.base() < x.base() is well-formed and convertible to bool.
Returns: !(y < x).

template<class Iterator1, class Iterator2>
constexpr bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Constraints:
x.base() < y.base() is well-formed and convertible to bool.
Returns: !(x < y).

template<class Iterator1, three_way_comparable_with<Iterator1> Iterator2>
constexpr compare_three_way_result_t<Iterator1, Iterator2>
operator<=>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Returns: x.base() <=> y.base().

23.5.3.9 Non-member functions

template<class Iterator1, class Iterator2>
constexpr auto operator-(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y)
-> decltype(x.base() - y.base());

template<sized_sentinel_for<Iterator> S>
friend constexpr iter_difference_t<Iterator>
operator-(const move_sentinel<S>& x, const move_iterator& y);

template<sized_sentinel_for<Iterator> S>
friend constexpr iter_difference_t<Iterator>
operator-(const move_iterator& x, const move_sentinel<S>& y);

Returns: x.base() - y.base().

template<class Iterator>
constexpr move_iterator<Iterator>
operator+(iter_difference_t<Iterator> n, const move_iterator<Iterator>& x);

Constraints: x + n is well-formed and has type Iterator.
Returns: x + n.

friend constexpr iter_rvalue_reference_t<Iterator>
iter_move(const move_iterator& i)
noexcept(noexcept(ranges::iter_move(i.current)));

Effects: Equivalent to: return ranges::iter_move(i.current);

template<indirectly_swappable<Iterator> Iterator2>
friend constexpr void
iter_swap(const move_iterator& x, const move_iterator<Iterator2>& y)
noexcept(noexcept(ranges::iter_swap(x.current, y.current)));

Effects: Equivalent to: ranges::iter_swap(x.current, y.current).

template<class Iterator>
constexpr move_iterator<Iterator>
make_move_iterator(Iterator i);

Returns: move_iterator<Iterator>(std::move(i)).

23.5.3.10 Class template move_sentinel

Class template move_sentinel is a sentinel adaptor useful for denoting ranges together with move_iterator. When an input iterator type I and sentinel type S model sentinel_for<S, I>, move_sentinel<S> and move_iterator<I> model sentinel_for<move_sentinel<S>, move_iterator<I>> as well.

[Example 1: A move_if algorithm is easily implemented with copy_if using move_iterator and move_sentinel:
template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, indirect Unary predicate<I> Pred>
requires indirectly movable<I, O>
void move_if(I first, S last, O out, Pred pred) {
  std::ranges::copy_if(move_iterator<I>{first}, move_sentinel<S>{last}, out, pred);
}

namespace std {
  template<semiregular S>
  class move_sentinel {
    public:
      constexpr move_sentinel();
      constexpr explicit move_sentinel(S s);
      template<class S2>
      requires convertible_to<const S2&, S>
      constexpr move_sentinel(const move_sentinel<S2>& s); 
      template<class S2>
      requires assignable_from<S&, const S2&> 
      constexpr move_sentinel& operator=(const move_sentinel<S2>& s);
      constexpr S base() const;
    private:
      S last;  // exposition only
    };
}

23.5.3.11 Operations [move.sent.ops]

constexpr move_sentinel();                           1
Effects: Value-initializes last. If is_trivially_default_constructible_v<S> is true, then this constructor is a constexpr constructor.

constexpr explicit move_sentinel(S s);               2
Effects: Initializes last with std::move(s).

template<class S2>
requires convertible_to<const S2&, S>
constexpr move_sentinel(const move_sentinel<S2>& s);  3
Effects: Initializes last with s.last.

template<class S2>
requires assignable_from<S&, const S2&>
constexpr move_sentinel& operator=(const move_sentinel<S2>& s);  4
Effects: Equivalent to: last = s.last; return *this;

constexpr S base() const;                           5
Returns: last.

23.5.4 Common Iterators [iterators.common]

23.5.4.1 Class template common_iterator [common.iterator]

Class template common_iterator is an iterator/sentinel adaptor that is capable of representing a non-common range of elements (where the types of the iterator and sentinel differ) as a common range (where they are the same). It does this by holding either an iterator or a sentinel, and implementing the equality comparison operators appropriately.

[Note 1: The common_iterator type is useful for interfacing with legacy code that expects the begin and end of a range to have the same type. — end note]

[Example 1:
  template<class ForwardIterator>
  void fun(ForwardIterator begin, ForwardIterator end);]
list<int> s;
// populate the list s
using CI = common_iterator<counted_iterator<int>::iterator, default_sentinel_t>;
// call fun on a range of 10 ints
fun(CI(counted_iterator(s.begin(), 10)), CI(default_sentinel));
—end example
namespace std {
    template<input_or_output_iterator I, sentinel_for<I> S>
    requires (!same_as<I, S> && copyable<I>)
    class common_iterator {
    public:
        constexpr common_iterator() = default;
        constexpr common_iterator(I i);
        constexpr common_iterator(S s);
        template<class I2, class S2>
        requires convertible_to<const I2&, I> && convertible_to<const S2&, S>
        constexpr common_iterator(const common_iterator<I2, S2>& x);
        template<class I2, class S2>
        requires convertible_to<const I2&, I> && convertible_to<const S2&, S> &&
                        assignable_from<I&, const I2&> && assignable_from<S&, const S2&>
        common_iterator& operator=(const common_iterator<I2, S2>& x);
        decltype(auto) operator*();
        decltype(auto) operator*() const
            requires dereferenceable<const I>;
        decltype(auto) operator->() const
            requires see below;
        common_iterator& operator++();
        decltype(auto) operator++(int);
        template<class I2, sentinel_for<I> S2>
        requires sentinel_for<S, I2>
        friend bool operator==(const common_iterator& x, const common_iterator<I2, S2>& y);
        template<class I2, sentinel_for<I> S2>
        requires sentinel_for<S, I2> && equality_comparable_with<I, I2>
        friend bool operator==(const common_iterator& x, const common_iterator<I2, S2>& y);
        template<sized_sentinel_for<I> I2, sized_sentinel_for<I> S2>
        requires sized_sentinel_for<S, I2>
        friend iter_difference_t<I2> operator-(
            const common_iterator& x, const common_iterator<I2, S2>& y);
        friend iter_rvalue_reference_t<I> iter_move(const common_iterator& i) noexcept(noexcept(ranges::iter_move(declval<const I&>())))
            requires input_iterator<I>;
        template<indirectly_swappable<I> I2, class S2>
        friend void iter_swap(const common_iterator& x, const common_iterator<I2, S2>& y)
            noexcept(noexcept(ranges::iter_swap(declval<const I&>(), declval<const I2&>())))
            ;
    private:
        variant<I, S> v_; // exposition only
    };
    template<class I, class S>
    struct incrementable_traits<common_iterator<I, S>> {
        using difference_type = iter_difference_t<I>;
    };

§ 23.5.4.1
template<input_iterator I, class S> struct iterator_traits<common_iterator<I, S>> {
    using iterator_concept = see below;
    using iterator_category = see below;
    using value_type = iter_value_t<I>;
    using difference_type = iter_difference_t<I>;
    using pointer = see below;
    using reference = iter_reference_t<I>;
};

23.5.4.2 Associated types [common.iter.types]

1 The nested typedef-names of the specialization of iterator_traits for common_iterator<I, S> are defined as follows.

1.1 — iterator_concept denotes forward_iterator_tag if I models forward_iterator; otherwise it denotes input_iterator_tag.

1.2 — iterator_category denotes forward_iterator_tag if iterator_traits<I>::iterator_category models derived_from<forward_iterator_tag>; otherwise it denotes input_iterator_tag.

1.3 — If the expression a.operator->() is well-formed, where a is an lvalue of type const common_iterator<I, S>, then pointer denotes the type of that expression. Otherwise, pointer denotes void.

23.5.4.3 Constructors and conversions [common.iter.const]

castexpr common_iterator(I i);

1 Effects: Initializes v_ as if by v_{in_place_type<I>, std::move(i)}.

castexpr common_iterator(S s);

2 Effects: Initializes v_ as if by v_{in_place_type<S>, std::move(s)}.

template<class I2, class S2>
requires convertible_to<const I2&, I> && convertible_to<const S2&, S>
    constexpr common_iterator(const common_iterator<I2, S2>& x);

3 Preconditions: x.v_.valueless_by_exception() is false.

4 Effects: Initializes v_ as if by v_{in_place_index<i>, get<i>(x.v_)}, where i is x.v_.index().

template<class I2, class S2>
requires convertible_to<const I2&, I> && convertible_to<const S2&, S> && assignable_from<I&, const I2&> && assignable_from<S&, const S2&>
    common_iterator& operator=(const common_iterator<I2, S2>& x);

5 Preconditions: x.v_.valueless_by_exception() is false.

6 Effects: Equivalent to:

6.1 — If v_.index() == x.v_.index(), then get<i>(v_) = get<i>(x.v_).

6.2 — Otherwise, v_.emplace<i>(get<i>(x.v_)).

where i is x.v_.index().

7 Returns: *this

23.5.4.4 Accessors [common.iter.access]

decltype(auto) operator*();
decltype(auto) operator*(const
requires dereferenceable<const I>;

1 Preconditions: holds_alternative<I>(v_).

2 Effects: Equivalent to: return *get<I>(v_);
decltype(auto) operator->() const
   requires see below;

3 The expression in the requires-clause is equivalent to:
   indirectly_readable<const I> &&
   (requires(const & i) { i.operator->(); } ||
    is_reference_v<iter_reference_t<I>> ||
    constructible_from<iter_value_t<I>, iter_reference_t<I>>)

   Preconditions: holds_alternative<I>(v_).

4 Effects:

   (5.1) — If I is a pointer type or if the expression get<I>(v_).operator->() is well-formed, equivalent to: return get<I>(v_);

   (5.2) — Otherwise, if iter_reference_t<I> is a reference type, equivalent to:
       auto& tmp = *get<I>(v_);
       return addressof(tmp);

   (5.3) — Otherwise, equivalent to: return proxy(*get<I>(v_)); where proxy is the exposition-only class:
       class proxy {
           iter_value_t<I> keep_
           proxy(iter_reference_t<I>&& x)
                : keep_(std::move(x)) {}  
           public:
               const iter_value_t<I>* operator->() const {
                   return addressof(keep_);
               }
       };

23.5.4.5 Navigation [common.iter.nav]

common_iterator& operator++();
1 Preconditions: holds_alternative<I>(v_).
2 Effects: Equivalent to ++get<I>(v_).
3 Returns: *this.

decaytype(auto) operator++(int);
4 Preconditions: holds_alternative<I>(v_).
5 Effects: If I models forward_iterator, equivalent to:
       common_iterator tmp = *this;
       +++this;
       return tmp;

   Otherwise, equivalent to: return get<I>(v_)++;

23.5.4.6 Comparisons [common.iter.cmp]

template<class I2, sentinel_for<I> S2>
   requires sentinel_for<S, I2>
friend bool operator==(   
    const common_iterator& x, const common_iterator<I2, S2>& y);
1 Preconditions: x.v_.valueless_by_exception() and y.v_.valueless_by_exception() are each false.
2 Returns: true if i == j, and otherwise get<i>(x.v_) == get<j>(y.v_), where i is x.v_.index() and j is y.v_.index().

template<class I2, sentinel_for<I> S2>
   requires sentinel_for<S, I2> & equality_comparable_with<I, I2>
friend bool operator==(   

§ 23.5.4.6
const common_iterator& x, const common_iterator<I2, S2>& y);

Preconditions: x.v_.valueless_by_exception() and y.v_.valueless_by_exception() are each false.

Returns: true if i and j are each 1, and otherwise \( \text{get<}i\text{>(x.v_)} = \text{get<}j\text{>(y.v_)} \), where i is x.v_.index() and j is y.v_.index().

template<sized_sentinel_for<I> I2, sized_sentinel_for<I> S2>
requires sized_sentinel_for<S, I2>
friend iter_difference_t<I2> operator-(
    const common_iterator& x, const common_iterator<I2, S2>& y);

Preconditions: x.v_.valueless_by_exception() and y.v_.valueless_by_exception() are each false.

Returns: 0 if i and j are each 1, and otherwise \( \text{get<}i\text{(x.v_)} - \text{get<}j\text{(y.v_)} \), where i is x.v_.index() and j is y.v_.index().

23.5.4.7 Customizations

friend iter_rvalue_reference_t<I> iter_move(const common_iterator& i)
    noexcept(noexcept(ranges::iter_move(declval<const I&>())))
    requires input_iterator<I>;

Preconditions: holds_alternative<I>(v_).

Effects: Equivalent to: return ranges::iter_move(get<I>(i.v_));

template<indirectly_swappable<I> I2, class S2>
friend void iter_swap(const common_iterator& x, const common_iterator<I2, S2>& y)
    noexcept(noexcept(ranges::iter_swap(declval<const I&>(), declval<const I2&>())));

Preconditions: holds_alternative<I>(x.v_) and holds_alternative<I2>(y.v_) are each true.

Effects: Equivalent to ranges::iter_swap(get<I>(x.v_), get<I2>(y.v_)).

23.5.5 Default sentinel

namespace std {
    struct default_sentinel_t { };  
}

Class default_sentinel_t is an empty type used to denote the end of a range. It can be used together with iterator types that know the bound of their range (e.g., counted_iterator (23.5.6.1)).

23.5.6 Counted iterators

23.5.6.1 Class template counted_iterator

Class template counted_iterator is an iterator adaptor with the same behavior as the underlying iterator except that it keeps track of the distance to the end of its range. It can be used together with default_sentinel in calls to generic algorithms to operate on a range of N elements starting at a given position without needing to know the end position a priori.

[Example 1:

list<string> s;
    // populate the list s with at least 10 strings
vector<string> v;
    // copies 10 strings into v:
    ranges::copy(counted_iterator(s.begin(), 10), default_sentinel, back_inserter(v));

—end example]

Two values i1 and i2 of types counted_iterator<I1> and counted_iterator<I2> refer to elements of the same sequence if and only if next(i1.base(), i1.count()) and next(i2.base(), i2.count()) refer to the same (possibly past-the-end) element.
namespace std {
    template<input_or_output_iterator I>
    class counted_iterator {
public:
    using iterator_type = I;

    constexpr counted_iterator() = default;
    constexpr counted_iterator(I x, iter_difference_t<I> n);
    template<class I2>
    requires convertible_to<const I2&, I>
    constexpr counted_iterator(const counted_iterator<I2>& x);
    template<class I2>
    requires assignable_from<I, const I2&>
    constexpr counted_iterator& operator=(const counted_iterator<I2>& x);

    constexpr I base() const & requires copy_constructible<I>;
    constexpr I base() &&;
    constexpr iter_difference_t<I> count() const noexcept;
    constexpr decltype(auto) operator*();
    constexpr decltype(auto) operator*() const requires dereferenceable<const I>;
    constexpr counted_iterator& operator++();
    decltype(auto) operator++(int);
    constexpr counted_iterator operator++(int) requires forward_iterator<I>;
    constexpr counted_iterator operator--();
    constexpr counted_iterator operator--(int) requires bidirectional_iterator<I>;
    constexpr counted_iterator operator+(iter_difference_t<I> n) const requires random_access_iterator<I>;
    friend constexpr counted_iterator operator+(iter_difference_t<I> n, const counted_iterator& x) requires random_access_iterator<I>;
    constexpr counted_iterator& operator+=(iter_difference_t<I> n) requires random_access_iterator<I>;
    constexpr counted_iterator operator-(iter_difference_t<I> n) const requires random_access_iterator<I>;
    template<common_with<I> I2>
    friend constexpr iter_difference_t<I2> operator- (const counted_iterator& x, const counted_iterator<I2>& y);
    constexpr counted_iterator& operator-=(iter_difference_t<I> n) requires random_access_iterator<I>;
    constexpr decltype(auto) operator[](iter_difference_t<I> n) const requires random_access_iterator<I>;
    template<common_with<I> I2>
    friend constexpr bool operator== (const counted_iterator& x, const counted_iterator<I2>& y);
    template<common_with<I> I2>
    friend constexpr strong_ordering operator<=> (const counted_iterator& x, const counted_iterator<I2>& y);

    constexpr counted_iterator& operator++();
    decltype(auto) operator++(int);
    constexpr counted_iterator operator++(int) requires forward_iterator<I>;
    constexpr counted_iterator operator--();
    constexpr counted_iterator operator--(int) requires bidirectional_iterator<I>;
    constexpr counted_iterator operator+(iter_difference_t<I> n) const requires random_access_iterator<I>;
    friend constexpr counted_iterator operator+(iter_difference_t<I> n, const counted_iterator& x) requires random_access_iterator<I>;
    constexpr counted_iterator& operator+=(iter_difference_t<I> n) requires random_access_iterator<I>;
    constexpr counted_iterator operator-(iter_difference_t<I> n) const requires random_access_iterator<I>;
    template<common_with<I> I2>
    friend constexpr iter_difference_t<I2> operator- (const counted_iterator& x, const counted_iterator<I2>& y);
    constexpr counted_iterator& operator-=(iter_difference_t<I> n) requires random_access_iterator<I>;
    constexpr decltype(auto) operator[](iter_difference_t<I> n) const requires random_access_iterator<I>;
    template<common_with<I> I2>
    friend constexpr bool operator== (const counted_iterator& x, const counted_iterator<I2>& y);
    template<common_with<I> I2>
    friend constexpr strong_ordering operator<=> (const counted_iterator& x, const counted_iterator<I2>& y);

§ 23.5.6.1 969
friend constexpr iter_rvalue_reference_t<I> iter_move(const counted_iterator& i)
    noexcept(noexcept(ranges::iter_move(i.current)))
    requires input_iterator<I>;
//template<indirectly_swappable<I> I2>
friend constexpr void iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
    noexcept(noexcept(ranges::iter_swap(x.current, y.current)));

private:
    I current = I();  // exposition only
    iter_difference_t<I> length = 0;  // exposition only
};

template<class I>
struct incrementable_traits<counted_iterator<I>> {
    using difference_type = iter_difference_t<I>;
};

template<input_iterator I>
struct iterator_traits<counted_iterator<I>> : iterator_traits<I> {
    using pointer = void;
};

23.5.6.2 Constructors and conversions

constexpr counted_iterator(I i, iter_difference_t<I> n);
1  Preconditions: n >= 0.
2  Effects: Initializes current with std::move(i) and length with n.

template<class I2>
    requires convertible_to<const I2&, I>
constexpr counted_iterator(const counted_iterator<I2>& x);
3  Effects: Initializes current with x.current and length with x.length.

template<class I2>
    requires assignable_from<I&, const I2&>
constexpr counted_iterator& operator=(const counted_iterator<I2>& x);
4  Effects: Assigns x.current to current and x.length to length.
5  Returns: *this.

23.5.6.3 Accessors

constexpr I base() const & requires copy_constructible<I>;
1  Effects: Equivalent to: return current;

constexpr I base() &&;
2  Returns: std::move(current).

constexpr iter_difference_t<I> count() const noexcept;
3  Effects: Equivalent to: return length;

23.5.6.4 Element access

castexpr decltype(auto) operator*();
constexpr decltype(auto) operator*() const
    requires dereferenceable<const I>;
1  Effects: Equivalent to: return *current;

castexpr decltype(auto) operator[](iter_difference_t<I> n) const
    requires random_access_iterator<I>;
2  Preconditions: n < length.
3 Effects: Equivalent to: return current[n];

23.5.6.5 Navigation [counted.iter.nav]

constexpr counted_iterator& operator++();

1 Preconditions: length > 0.

2 Effects: Equivalent to:
    ++current;
    --length;
    return *this;

decltype(auto) operator++(int);

3 Preconditions: length > 0.

4 Effects: Equivalent to:
    --length;
    try { return current++; } 
    catch(...) { ++length; throw; }

constexpr counted_iterator operator++(int)
    requires forward_iterator<I>;

5 Effects: Equivalent to:
    counted_iterator tmp = *this;
    +++this;
    return tmp;

constexpr counted_iterator& operator--();
    requires bidirectional_iterator<I>;

6 Effects: Equivalent to:
    --current;
    +++length;
    return *this;

constexpr counted_iterator& operator--(int)
    requires bidirectional_iterator<I>;

7 Effects: Equivalent to:
    counted_iterator tmp = *this;
    --*this;
    return tmp;

constexpr counted_iterator operator+(iter_difference_t<I> n) const
    requires random_access_iterator<I>;

8 Effects: Equivalent to: return counted_iterator(current + n, length - n);

friend constexpr counted_iterator operator+(
    iter_difference_t<I> n, const counted_iterator& x)
    requires random_access_iterator<I>;

9 Effects: Equivalent to: return x + n;

constexpr counted_iterator& operator+=(iter_difference_t<I> n)
    requires random_access_iterator<I>;

10 Preconditions: n <= length.
11 Effects: Equivalent to:
    current += n;
    length -= n;
    return *this;

§ 23.5.6.5
constexpr counted_iterator operator-(iter_difference_t<I> n) const
requires random_access_iterator<I>;

Effects: Equivalent to: return counted_iterator(current - n, length + n);

template<common_with<I> I2>
friend constexpr iter_difference_t<I2> operator-(
    const counted_iterator& x, const counted_iterator<I2>& y);

Preconditions: x and y refer to elements of the same sequence (23.5.6.1).

Effects: Equivalent to: return y.length - x.length;

friend constexpr iter_difference_t<I> operator-=(
    const counted_iterator& x, default_sentinel_t);

Effects: Equivalent to: return -x.length;

friend constexpr iter_difference_t<I> operator-=(
    const counted_iterator& x, default_sentinel_t, const counted_iterator& y);

Effects: Equivalent to: return y.length;

constexpr counted_iterator& operator-=(iter_difference_t<I> n)
requires random_access_iterator<I>;

Preconditions: -n <= length.

Effects: Equivalent to:
    current -= n;
    length += n;
    return *this;

23.5.6.6 Comparisons [counted.iter.cmp]

template<common_with<I> I2>
friend constexpr bool operator==(const counted_iterator& x, const counted_iterator<I2>& y);

Preconditions: x and y refer to elements of the same sequence (23.5.6.1).

Effects: Equivalent to: return x.length == y.length;

friend constexpr bool operator==(const counted_iterator& x, default_sentinel_t);

Effects: Equivalent to: return x.length == 0;

template<common_with<I> I2>
friend constexpr strong_ordering operator<>(
    const counted_iterator& x, const counted_iterator<I2>& y);

Preconditions: x and y refer to elements of the same sequence (23.5.6.1).

Effects: Equivalent to: return y.length <=> x.length;

[Note 1: The argument order in the Effects: element is reversed because length counts down, not up. — end note]

23.5.6.7 Customizations [counted.iter.cust]

friend constexpr iter_rvalue_reference_t<I>
    iter_move(const counted_iterator& i)
requires input_iterator<I>;

Effects: Equivalent to: return ranges::iter_move(i.current);

template<indirectly_swappable<I> I2>
friend constexpr void
    iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
requires input_iterator<I>;

Effects: Equivalent to ranges::iter_swap(x.current, y.current).
23.5.7 Unreachable sentinel

Class unreachable_sentinel_t can be used with any weakly_incrementable type to denote the “upper bound” of an unbounded interval.

Example 1:
```c
char* p;
// set p to point to a character buffer containing newlines
char* nl = find(p, unreachable_sentinel, '\n');
```

Provided a newline character really exists in the buffer, the use of unreachable_sentinel above potentially makes the call to find more efficient since the loop test against the sentinel does not require a conditional branch. — end example]

namespace std {
    struct unreachable_sentinel_t {
        template<weakly_incrementable I>
        friend constexpr bool operator==(unreachable_sentinel_t, const I&) noexcept
            { return false; }
    }
}

23.6 Stream iterators

23.6.1 General

To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

Example 1:
```c
partial_sum(istream_iterator<double, char>(cin),
            istream_iterator<double, char>(),
            ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating-point numbers from cin, and prints the partial sums onto cout. — end example]

23.6.2 Class template istream_iterator

23.6.2.1 General

The class template istream_iterator is an input iterator (23.3.5.3) that reads successive elements from the input stream for which it was constructed.

namespace std {
    template<class T, class charT = char, class traits = char_traits<charT>,
             class Distance = ptrdiff_t>
    class istream_iterator {
        public:
            using iterator_category = input_iterator_tag;
            using value_type = T;
            using difference_type = Distance;
            using pointer = const T*;
            using reference = const T&;
            using char_type = charT;
            using traits_type = traits;
            using istream_type = basic_istream<charT,traits>;

            constexpr istream_iterator();
            constexpr istream_iterator(default_sentinel_t);
            istream_iterator(istream_type& s);
            istream_iterator(const istream_iterator& x) = default;
            ~istream_iterator() = default;
            istream_iterator& operator=(const istream_iterator&) = default;

            const T& operator*() const;
            const T* operator->() const;
            istream_iterator& operator++();
            istream_iterator operator++(int);
    }
friend bool operator==(const istream_iterator& i, default_sentinel_t);

private:
    basic_istream<charT,traits>* in_stream; // exposition only
    T value; // exposition only
};

2 The type T shall meet the Cpp17DefaultConstructible, Cpp17CopyConstructible, and Cpp17CopyAssignable requirements.

23.6.2.2 Constructors and destructor

constexpr istream_iterator();
constexpr istream_iterator(default_sentinel_t);

1 Effects: Constructs the end-of-stream iterator, value-initializing value.

2 Postconditions: in_stream == nullptr is true.

3 Remarks: If the initializer T() in the declaration auto x = T(); is a constant initializer (7.7), then these constructors are constexpr constructors.

istream_iterator(istream_type& s);

4 Effects: Initializes in_stream with addressof(s), value-initializes value, and then calls operator++().

5 Postconditions: in_stream == x.in_stream is true.

6 Remarks: If is_trivially_copy_constructible_v<T> is true, then this constructor is trivial.

~istream_iterator() = default;
7 Remarks: If is_trivially_destructible_v<T> is true, then this destructor is trivial.

23.6.2.3 Operations

const T& operator*() const;

1 Preconditions: in_stream != nullptr is true.

2 Returns: value.

const T* operator->() const;

3 Preconditions: in_stream != nullptr is true.

4 Returns: addressof(value).

istream_iterator& operator++();

5 Preconditions: in_stream != nullptr is true.

6 Effects: Equivalent to:
    if (!(*in_stream >> value))
        in_stream = nullptr;

7 Returns: *this.

istream_iterator operator++(int);

8 Preconditions: in_stream != nullptr is true.

9 Effects: Equivalent to:
    istream_iterator tmp = *this;
    ++*this;
    return tmp;

template<class T, class charT, class traits, class Distance>
bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
                const istream_iterator<T,charT,traits,Distance>& y);

10 Returns: x.in_stream == y.in_stream.
friend bool operator==(const istream_iterator& i, default_sentinel_t);

Returns: !i.in_stream.

23.6.3 Class template ostream_iterator

23.6.3.1 General

ostream_iterator writes (using operator<<) successive elements onto the output stream from which it was constructed. If it was constructed with charT* as a constructor argument, this string, called a delimiter string, is written to the stream after every T is written.

```cpp
namespace std {
    template<class T, class charT = char, class traits = char_traits<charT>>
    class ostream_iterator {
        public:
            using iterator_category = output_iterator_tag;
            using value_type = void;
            using difference_type = ptrdiff_t;
            using pointer = void;
            using reference = void;
            using char_type = charT;
            using traits_type = traits;
            using ostream_type = basic_ostream<charT,traits>;

            constexpr ostream_iterator() noexcept = default;
            ostream_iterator(ostream_type& s);
            ostream_iterator(ostream_type& s, const charT* delimiter);
            ostream_iterator(const ostream_iterator& x);
            ~ostream_iterator();
            ostream_iterator& operator=(const ostream_iterator&);
            ostream_iterator& operator=(const T& value);
            ostream_iterator& operator*();
            ostream_iterator& operator++();
            ostream_iterator& operator++(int);

            private:
                basic_ostream<charT,traits>* out_stream = nullptr; // exposition only
                const charT* delim = nullptr; // exposition only
        }
    }
}
```

23.6.3.2 Constructors and destructor

1. Effects: Initializes out_stream with addressof(s) and delim with nullptr.

2. Effects: Initializes out_stream with addressof(s) and delim with delimiter.

23.6.3.3 Operations

1. Effects: As if by:
   *out_stream << value;
   if (delim)
       *out_stream << delim;
   return *this;

2. Returns: *this.
23.6.4 Class template istreambuf_iterator

23.6.4.1 General

The class template `istreambuf_iterator` defines an input iterator (23.3.5.3) that reads successive characters from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (`streambuf_type::sgetc()` returns `traits::eof()`), the iterator becomes equal to the `end-of-stream` iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(nullptr)` both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a `constexpr` default constructor, and a trivial destructor.

The result of `operator*()` on an end-of-stream iterator is undefined. For any other iterator value a `char_type` value is returned. It is impossible to assign a character via an input iterator.

```cpp
namespace std {

  template<class charT, class traits = char_traits<charT>>
  class istreambuf_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type = charT;
    using difference_type = typename traits::off_type;
    using pointer = unspecified;
    using reference = charT;
    using char_type = charT;
    using traits_type = traits;
    using int_type = typename traits::int_type;
    using streambuf_type = basic_streambuf<charT,traits>;
    using istream_type = basic_istream<charT,traits>;

    class proxy;  // exposition only

    constexpr istreambuf_iterator() noexcept;
    constexpr istreambuf_iterator(default_sentinel_t) noexcept;
    istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
    ~istreambuf_iterator() = default;
    istreambuf_iterator(istream_type& s) noexcept;
    istreambuf_iterator(streambuf_type* s) noexcept;
    istreambuf_iterator(const proxy& p) noexcept;
    istreambuf_iterator& operator=(const istreambuf_iterator&) noexcept = default;
    charT operator*() const;
    istreambuf_iterator& operator++();
    proxy operator++(int);
    bool equal(const istreambuf_iterator& b) const;
    friend bool operator==(const istreambuf_iterator& i, default_sentinel_t s);

    private:
      streambuf_type* sbuf_;  // exposition only
    }

  } // namespace std
```

23.6.4.2 Class istreambuf_iterator::::proxy

Class `istreambuf_iterator<charT,traits>::::proxy` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class `istreambuf_iterator<charT,traits>::::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator*`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.
namespace std {
    template<class charT, class traits>
    class istreambuf_iterator<charT, traits>::proxy { // exposition only
      charT keep_;  // exposition only
      basic_streambuf<charT,traits>* sbuf_;  // exposition only
      proxy(charT c, basic_streambuf<charT,traits>* sbuf)  
        : keep_(c), sbuf_(sbuf) { }
    public:
        charT operator*() { return keep_; }
    }
}

23.6.4.3 Constructors [istreambuf.iterator.cons]

For each istreambuf_iterator constructor in this subclause, an end-of-stream iterator is constructed if and only if the exposition-only member sbuf_ is initialized with a null pointer value.

constexpr istreambuf_iterator() noexcept;
constexpr istreambuf_iterator(default_sentinel_t) noexcept;

Effects: Initializes sbuf_ with nullptr.

istreambuf_iterator(istream_type& s) noexcept;

Effects: Initializes sbuf_ with s.rdbuf().

istreambuf_iterator(streambuf_type* s) noexcept;

Effects: Initializes sbuf_ with s.

istreambuf_iterator(const proxy& p) noexcept;

Effects: Initializes sbuf_ with p.sbuf_.

23.6.4.4 Operations [istreambuf.iterator.ops]

charT operator*() const;

Returns: The character obtained via the streambuf member sbuf_->sgetc().

istreambuf_iterator& operator++();

Effects: As if by sbuf_->sbumpc().

Returns: *this.

proxy operator++(int);

Returns: proxy(sbuf_->sbumpc(), sbuf_).

bool equal(const istreambuf_iterator& b) const;

Returns: true if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what streambuf object they use.

template<class charT, class traits>  // exposition only
bool operator==(const istreambuf_iterator<charT,traits>& a, const istreambuf_iterator<charT,traits>& b);

Returns: a.equal(b).

friend bool operator==(const istreambuf_iterator& i, default_sentinel_t s);

Returns: i.equal(s).

23.6.5 Class template ostreambuf_iterator [ostreambuf.iterator]

23.6.5.1 General [ostreambuf.iterator.general]

The class template ostreambuf_iterator writes successive characters onto the output stream from which it was constructed.

§ 23.6.5.1
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class ostreambuf_iterator {
public:
    using iterator_category = output_iterator_tag;
    using value_type = void;
    using difference_type = ptrdiff_t;
    using pointer = void;
    using reference = void;
    using char_type = charT;
    using traits_type = traits;
    using streambuf_type = basic_streambuf<charT,traits>;
    using ostream_type = basic_ostream<charT,traits>;

    constexpr ostreambuf_iterator() noexcept = default;
    ostreambuf_iterator(ostream_type& s) noexcept;
    ostreambuf_iterator(streambuf_type* s) noexcept;

    ostreambuf_iterator& operator=(charT c);
    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);

    bool failed() const noexcept;

private:
    streambuf_type* sbuf_ = nullptr; // exposition only
    };
}

23.6.5.2 Constructors

ostreambuf_iterator(ostream_type& s) noexcept;

1 Preconditions: s.rdbuf() is not a null pointer.
2 Effects: Initializes sbuf_ with s.rdbuf().

ostreambuf_iterator(streambuf_type* s) noexcept;

3 Preconditions: s is not a null pointer.
4 Effects: Initializes sbuf_ with s.

23.6.5.3 Operations

ostreambuf_iterator& operator=(charT c);

1 Effects: If failed() yields false, calls sbuf_->sputc(c); otherwise has no effect.
2 Returns: *this.

ostreambuf_iterator& operator*();

3 Returns: *this.

ostreambuf_iterator& operator++();
ostreambuf_iterator& operator++(int);

4 Returns: *this.

bool failed() const noexcept;

5 Returns: true if in any prior use of member operator=, the call to sbuf_->sputc() returned
traits::eof(); or false otherwise.

23.7 Range access

In addition to being available via inclusion of the <iterator> header, the function templates in 23.7 are
available when any of the following headers are included: <array> (22.3.2), <deque> (22.3.3), <forward_list>
(22.3.4), <list> (22.3.5), <map> (22.4.2), <regex> (30.3), <set> (22.4.3), <span> (22.7.2), <string> (21.3.2),
<string_view> (21.4.2), <unordered_map> (22.5.2), <unordered_set> (22.5.3), and <vector> (22.3.6). Each of these templates is a designated customization point (16.4.5.2.1).

```cpp
template<class C> constexpr auto begin(C& c) -> decltype(c.begin());
template<class C> constexpr auto begin(const C& c) -> decltype(c.begin());
Returns: c.begin().
```

```cpp
template<class C> constexpr auto end(C& c) -> decltype(c.end());
template<class C> constexpr auto end(const C& c) -> decltype(c.end());
Returns: c.end().
```

```cpp
template<class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
Returns: array.
```

```cpp
template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c))) -> decltype(std::begin(c));
Returns: std::begin(c).
```

```cpp
template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c))) -> decltype(std::end(c));
Returns: std::end(c).
```

```cpp
template<class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin());
template<class C> constexpr auto rbegin(const C& c) -> decltype(c.rbegin());
Returns: c.rbegin().
```

```cpp
template<class C> constexpr auto rend(C& c) -> decltype(c.rend());
template<class C> constexpr auto rend(const C& c) -> decltype(c.rend());
Returns: c.rend().
```

```cpp
template<class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);
Returns: reverse_iterator<T*>(array + N).
```

```cpp
template<class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);
Returns: reverse_iterator<T*>(array).
```

```cpp
template<class E> constexpr reverse_iterator<const E*> rbegin(initializer_list<E> il);
Returns: reverse_iterator<const E*>(il.end()).
```

```cpp
template<class E> constexpr reverse_iterator<const E*> rend(initializer_list<E> il);
Returns: reverse_iterator<const E*>(il.begin()).
```

```cpp
template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));
Returns: std::rbegin(c).
```

```cpp
template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));
Returns: std::rend(c).
```

```cpp
template<class C> constexpr auto size(const C& c) -> decltype(c.size());
Returns: c.size().
```

```cpp
template<class T, size_t N> constexpr size_t size(const T (&array)[N]) noexcept;
Returns: N.
```

§ 23.7 979


-> common_type_t<ptrdiff_t, make_signed_t<decltype(c.size())>>;

Effects: Equivalent to:

return static_cast<common_type_t<ptrdiff_t, make_signed_t<decltype(c.size())>>>(c.size());

template<class T, ptrdiff_t N> constexpr ptrdiff_t ssize(const T (&array)[N]) noexcept;

Returns: N.

template<class C> [[nodiscard]] constexpr auto empty(const C& c) -> decltype(c.empty());

Returns: c.empty().

template<class T, size_t N> [[nodiscard]] constexpr bool empty(const T (&array)[N]) noexcept;

Returns: false.

template<class E> [[nodiscard]] constexpr bool empty(initializer_list<E> il) noexcept;

Returns: il.size() == 0.

template<class C> constexpr auto data(C& c) -> decltype(c.data());

template<class C> constexpr auto data(const C& c) -> decltype(c.data());

Returns: c.data().

template<class T, size_t N> constexpr T* data(T (&array)[N]) noexcept;

Returns: array.

template<class E> constexpr const E* data(initializer_list<E> il) noexcept;

Returns: il.begin().
24 Ranges library

24.1 General

This Clause describes components for dealing with ranges of elements.

The following subclauses describe range and view requirements, and components for range primitives as summarized in Table 90.

Table 90: Ranges library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.3</td>
<td>&lt;ranges&gt;</td>
</tr>
<tr>
<td>24.4</td>
<td>Requirements</td>
</tr>
<tr>
<td>24.5</td>
<td>Range utilities</td>
</tr>
<tr>
<td>24.6</td>
<td>Range factories</td>
</tr>
<tr>
<td>24.7</td>
<td>Range adaptors</td>
</tr>
</tbody>
</table>

24.2 Header <ranges> synopsis

```cpp
#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.2
#include <iterator> // see 23.2

namespace std::ranges {
inline namespace unspecified {
    // 24.3, range access
    inline constexpr unspecified begin = unspecified;
    inline constexpr unspecified end = unspecified;
    inline constexpr unspecified cbegin = unspecified;
    inline constexpr unspecified cend = unspecified;
    inline constexpr unspecified rbegin = unspecified;
    inline constexpr unspecified rend = unspecified;
    inline constexpr unspecified crbegin = unspecified;
    inline constexpr unspecified crend = unspecified;
    inline constexpr unspecified size = unspecified;
    inline constexpr unspecified ssize = unspecified;
    inline constexpr unspecified empty = unspecified;
    inline constexpr unspecified data = unspecified;
    inline constexpr unspecified cdata = unspecified;
}

// 24.4.2, ranges
template<class T> concept range = see below;

template<class T>
inline constexpr bool enable_borrowed_range = false;

template<class T>
concept borrowed_range = see below;

template<class T>
using iterator_t = decltype(ranges::begin(declval<T&>()));
template<class R>
using sentinel_t = decltype(ranges::end(declval<R&>()));
template<class R>
using range_difference_t = iter_difference_t<iterator_t<R>>;
```
```cpp
// 24.4.3, sized ranges
template<class>
inline constexpr bool disable_sized_range = false;

template<class T>
concept sized_range = see below;

// 24.4.4, views
template<class T>
inline constexpr bool enable_view = see below;

struct view_base { };

template<class T>
concept view = see below;

// 24.4.5, other range refinements
template<class R, class T>
concept output_range = see below;

template<class T>
concept input_range = see below;

template<class T>
concept forward_range = see below;

template<class T>
concept bidirectional_range = see below;

template<class T>
concept random_access_range = see below;

template<class T>
concept contiguous_range = see below;

template<class T>
concept common_range = see below;

template<class T>
concept viewable_range = see below;

// 24.5.3, class template view_interface
template<class D>
requires is_class_v<D> && same_as<D, remove_cv_t<D>>
class view_interface;

// 24.5.4, sub-ranges
enum class subrange_kind : bool { unsized, sized };

template<input_or_output_iterator I, sentinel_for<I> S = I, subrange_kind K = see below>
requires (K == subrange_kind::sized || !sized_sentinel_for<S, I>)
class subrange;

template<input_or_output_iterator I, sentinel_for<I> S, subrange_kind K>
inline constexpr bool enable_borrowed_range<subrange<I, S, K>> = true;
```

§ 24.2 982
// 24.5.5, dangling iterator handling
struct dangling;

template<range R>
using borrowed_iterator_t = conditional_t<borrowed_range<R>, iterator_t<R>, dangling>;

template<range R>
using borrowed subrange_t = conditional_t<borrowed range<R>, subrange<iterator_t<R>>, dangling>;

// 24.6.2, empty view
template<class T>
requires is_object_v<T>
class empty_view;

template<class T>
inline constexpr bool enable_borrowed_range<empty_view<T>> = true;

namespace views {
    template<class T>
    inline constexpr empty_view<T> empty{};
}

// 24.6.3, single view
template<copy_constructible T>
requires is_object_v<T>
class single_view;

namespace views { inline constexpr unspecified single = unspecified; }

// 24.6.4, iota view
template<weakly_incrementable W, semiregular Bound = unreachable_sentinel_t>
requires weakly-equality-comparable-with<W, Bound> && semiregular<W>
class iota_view;

template<weakly_incrementable W, semiregular Bound>
inline constexpr bool enable_borrowed_range<iota_view<W, Bound>> = true;

namespace views { inline constexpr unspecified iota = unspecified; }

// 24.6.5, istream view
template<movable Val, class CharT, class Traits = char_traits<CharT>>
requires see below
class basic_istream_view;

template<class Val, class CharT, class Traits>
basic_istream_view<Val, CharT, Traits> istream_view(basic_istream<CharT, Traits>& s);

// 24.7.4, all view
namespace views {
    inline constexpr unspecified all = unspecified;

    template<viewable_range R>
    using all_t = decltype(all(declval<R>()));
}

template<range R>
requires is_object_v<R>
class ref_view;

template<class T>
inline constexpr bool enable_borrowed_range<ref_view<T>> = true;
// 24.7.5, filter view
template<input_range V, indirect_unary_predicate<iterator_t<V>> Pred>
    requires view<V> && is_object_v<Pred>
    class filter_view;

namespace views { inline constexpr unspecified filter = unspecified; }

// 24.7.6, transform view
template<input_range V, copy_constructible F>
    requires view<V> && is_object_v<F> &&
        regular_invocable<F&, range_reference_t<V>>, &&
        can_reference<invoke_result_t<F&, range_reference_t<V>>>;
    class transform_view;

namespace views { inline constexpr unspecified transform = unspecified; }

// 24.7.7, take view
template<view> class take_view;

namespace views { inline constexpr unspecified take = unspecified; }

// 24.7.8, take while view
template<view V, class Pred>
    requires input_range<V> && is_object_v<Pred> &&
        indirect_unary_predicate<const Pred, iterator_t<V>>;
    class take_while_view;

namespace views { inline constexpr unspecified take_while = unspecified; }

// 24.7.9, drop view
template<view V>
    class drop_view;

namespace views { inline constexpr unspecified drop = unspecified; }

// 24.7.10, drop while view
template<view V, class Pred>
    requires input_range<V> && is_object_v<Pred> &&
        indirect_unary_predicate<const Pred, iterator_t<V>>;
    class drop_while_view;

namespace views { inline constexpr unspecified drop_while = unspecified; }

// 24.7.11, join view
template<input_range V>
    requires view<V> && input_range<range_reference_t<V>> &&
    (is_reference_v<range_reference_t<V>> ||
     view<range_value_t<V>>);
    class join_view;

namespace views { inline constexpr unspecified join = unspecified; }

// 24.7.12, split view
template<class R>
    concept tiny-range = see below; // exposition only

template<input_range V, forward_range Pattern>
    requires view<V> && view<Pattern> &&
        indirectly_comparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to> &&
        (forward_range<V> || tiny_range<Pattern>);
    class split_view;

namespace views { inline constexpr unspecified split = unspecified; }
// 24.7.13, counted view
namespace views { inline constexpr unspecified counted = unspecified; }

// 24.7.14, common view
template<view V>
  requires (!common_range<V> && copyable<iterator_t<V>>)
class common_view;

namespace views { inline constexpr unspecified common = unspecified; }

// 24.7.15, reverse view
template<view V>
  requires bidirectional_range<V>
class reverse_view;

namespace views { inline constexpr unspecified reverse = unspecified; }

// 24.7.16, elements view
template<input_range V, size_t N>
  requires see below
class elements_view;

template<class R>
  using keys_view = elements_view<views::all_t<R>, 0>;
template<class R>
  using values_view = elements_view<views::all_t<R>, 1>;

namespace views {
  template<size_t N>
    inline constexpr unspecified elements = unspecified;
  inline constexpr auto keys = elements<0>;
  inline constexpr auto values = elements<1>;
}

namespace std {
  namespace views = ranges::views;

  template<class I, class S, ranges::subrange_kind K>
  struct tuple_size<ranges::subrange<I, S, K>>
    : integral_constant<size_t, 2> {};
  template<class I, class S, ranges::subrange_kind K>
  struct tuple_element<0, ranges::subrange<I, S, K>> {
    using type = I;
  };
  template<class I, class S, ranges::subrange_kind K>
  struct tuple_element<1, ranges::subrange<I, S, K>> {
    using type = S;
  };
  template<class I, class S, ranges::subrange_kind K>
  struct tuple_element<0, const ranges::subrange<I, S, K>> {
    using type = I;
  };
  template<class I, class S, ranges::subrange_kind K>
  struct tuple_element<1, const ranges::subrange<I, S, K>> {
    using type = S;
  };
}

1 Within this Clause, for an integer-like type X (23.3.4.4), make_unsigned-like-t<X> denotes make_unsigned-t<X> if X is an integer type; otherwise, it denotes a corresponding unspecified unsigned-integer-like type of the same width as X. For an expression x of type X, to_unsigned-like(x) is x explicitly converted to make_unsigned-like-t<X>.
24.3 Range access

24.3.1 General

In addition to being available via inclusion of the `<ranges>` header, the customization point objects in 24.3 are available when `<iterator>` (23.2) is included.

Within 24.3, the reified object of a subexpression E denotes

1. the same object as E if E is a glvalue, or
2. the result of applying the temporary materialization conversion (7.3.5) to E otherwise.

24.3.2 ranges::begin

1. The name ranges::begin denotes a customization point object (16.3.3.3.6).
2. Given a subexpression E with type T, let t be an lvalue that denotes the reified object for E. Then:
   1. If E is an rvalue and enable_borrowed_range<remove_cv_t<T>> is false, ranges::begin(E) is ill-formed.
   2. Otherwise, if T is an array type (6.8.3) and remove_all_extents_t<T> is an incomplete type, ranges::begin(E) is ill-formed with no diagnostic required.
   3. Otherwise, if T is an array type, ranges::begin(E) is expression-equivalent to t + 0.
   4. Otherwise, if decay-copy(t.begin()) is a valid expression whose type models input_or_output_iterator, ranges::begin(E) is expression-equivalent to decay-copy(t.begin()).
   5. Otherwise, if T is a class or enumeration type and decay-copy(begin(t)) is a valid expression whose type models input_or_output_iterator with overload resolution performed in a context in which unqualified lookup for begin finds only the declarations
      
      ```
      void begin(auto&) = delete;
      void begin(const auto&) = delete;
      ```

      then ranges::begin(E) is expression-equivalent to decay-copy(begin(t)) with overload resolution performed in the above context.
   6. Otherwise, ranges::begin(E) is ill-formed.

[Note 1: Diagnosable ill-formed cases above result in substitution failure when ranges::begin(E) appears in the immediate context of a template instantiation. — end note]

3. [Note 2: Whenever ranges::begin(E) is a valid expression, its type models input_or_output_iterator. — end note]

24.3.3 ranges::end

1. The name ranges::end denotes a customization point object (16.3.3.3.6).
2. Given a subexpression E with type T, let t be an lvalue that denotes the reified object for E. Then:
   1. If E is an rvalue and enable_borrowed_range<remove_cv_t<T>> is false, ranges::end(E) is ill-formed.
   2. Otherwise, if T is an array type (6.8.3) and remove_all_extents_t<T> is an incomplete type, ranges::end(E) is ill-formed with no diagnostic required.
   3. Otherwise, if T is an array of unknown bound, ranges::end(E) is ill-formed.
   4. Otherwise, if T is an array, ranges::end(E) is expression-equivalent to t + extent_v<T>.
   5. Otherwise, if decay-copy(t.end()) is a valid expression whose type models sentinel_for<iterator_t<T>> then ranges::end(E) is expression-equivalent to decay-copy(t.end()).
   6. Otherwise, if T is a class or enumeration type and decay-copy(end(t)) is a valid expression whose type models sentinel_for<iterator_t<T>> with overload resolution performed in a context in which unqualified lookup for end finds only the declarations
      
      ```
      void end(auto&) = delete;
      void end(const auto&) = delete;
      ```

      then ranges::end(E) is expression-equivalent to decay-copy(end(t)) with overload resolution performed in the above context.
   7. Otherwise, ranges::end(E) is ill-formed.
24.3.4 ranges::cbegin

The name ranges::cbegin denotes a customization point object (16.3.3.3.6). The expression ranges::cbegin(E) for a subexpression E of type T is expression-equivalent to:

(1.1)  ranges::begin(static_cast<const T&>(E)) if E is an lvalue.
(1.2)  Otherwise, ranges::begin(static_cast<const T&&>(E)).

[Note 1: Whenever ranges::cbegin(E) is a valid expression, its type models input_or_output_iterator. — end note]

24.3.5 ranges::cend

The name ranges::cend denotes a customization point object (16.3.3.3.6). The expression ranges::cend(E) for a subexpression E of type T is expression-equivalent to:

(1.1)  ranges::end(static_cast<const T&>(E)) if E is an lvalue.
(1.2)  Otherwise, ranges::end(static_cast<const T&&>(E)).

[Note 1: Whenever ranges::cend(E) is a valid expression, the types S and I of the expressions ranges::cend(E) and ranges::cbegin(E) model sentinel_for<S, I>. — end note]

24.3.6 ranges::rbegin

The name ranges::rbegin denotes a customization point object (16.3.3.3.6).

Given a subexpression E with type T, let t be an lvalue that denotes the reified object for E. Then:

(2.1)  If E is an rvalue and enable_borrowed_range<remove_cv_t<T>> is false, ranges::rbegin(E) is ill-formed.
(2.2)  Otherwise, if T is an array type (6.8.3) and remove_all_extents_t<T> is an incomplete type, ranges::rbegin(E) is ill-formed with no diagnostic required.
(2.3)  Otherwise, if decay-copy(t.rbegin()) is a valid expression whose type models input_or_output_iterator with overload resolution performed in a context in which unqualified lookup for rbegin finds only the declarations
        void rbegin(auto&) = delete;
        void rbegin(const auto&) = delete;
    then ranges::rbegin(E) is expression-equivalent to decay-copy(rbegin(t)) with overload resolution performed in the above context.
(2.4)  Otherwise, if T is a class or enumeration type and decay-copy(rbegin(t)) is a valid expression whose type models input_or_output_iterator with overload resolution performed in a context in which unqualified lookup for rbegin finds only the declarations
        void rbegin(auto&) = delete;
        void rbegin(const auto&) = delete;
    then ranges::rbegin(E) is expression-equivalent to decay-copy(rbegin(t)) with overload resolution performed in the above context.
(2.5)  Otherwise, if both ranges::begin(t) and ranges::end(t) are valid expressions of the same type which models bidirectional_iterator (23.3.4.12), ranges::rbegin(E) is expression-equivalent to make_reverse_iterator(ranges::end(t)).
(2.6)  Otherwise, ranges::rbegin(E) is ill-formed.

[Note 1: Diagnosable ill-formed cases above result in substitution failure when ranges::rbegin(E) appears in the immediate context of a template instantiation. — end note]

24.3.7 ranges::rend

The name ranges::rend denotes a customization point object (16.3.3.3.6).

Given a subexpression E with type T, let t be an lvalue that denotes the reified object for E. Then:

(2.1)  If E is an rvalue and enable_borrowed_range<remove_cv_t<T>> is false, ranges::rend(E) is ill-formed.
Otherwise, if \( T \) is an array type (6.8.3) and \( \text{remove\_all\_extents\_t}<T> \) is an incomplete type, \( \text{ranges}::\text{rend}(E) \) is ill-formed with no diagnostic required.

Otherwise, if \( \text{decay\_copy}(t.\text{rend}()) \) is a valid expression whose type models \( \text{sentinel\_for}<\text{decl\_type}(\text{ranges}::\text{rbegin}(E))> \) then \( \text{ranges}::\text{rend}(E) \) is expression-equivalent to \( \text{decay\_copy}(t.\text{rend}()) \).

Otherwise, if \( T \) is a class or enumeration type and \( \text{decay\_copy}(\text{rend}(t)) \) is a valid expression whose type models \( \text{sentinel\_for}<\text{decl\_type}(\text{ranges}::\text{rbegin}(E))> \) with overload resolution performed in a context in which unqualified lookup for \( \text{rend} \) finds only the declarations

```cpp
void rend(auto&) = delete;
void rend(const auto&) = delete;
```

then \( \text{ranges}::\text{rend}(E) \) is expression-equivalent to \( \text{decay\_copy}(\text{rend}(t)) \) with overload resolution performed in the above context.

Otherwise, if both \( \text{ranges}::\text{begin}(t) \) and \( \text{ranges}::\text{end}(t) \) are valid expressions of the same type which models \( \text{bidirectional\_iterator} \) (23.3.4.12), then \( \text{ranges}::\text{rend}(E) \) is expression-equivalent to \( \text{make\_reverse\_iterator}(\text{ranges}::\text{begin}(t)) \).

Otherwise, \( \text{ranges}::\text{rend}(E) \) is ill-formed.

3 [Note 1: Diagnosable ill-formed cases above result in substitution failure when \( \text{ranges}::\text{rend}(E) \) appears in the immediate context of a template instantiation. — end note]

4 [Note 2: Whenever \( \text{ranges}::\text{rend}(E) \) is a valid expression, the types \( S \) and \( I \) of the expressions \( \text{ranges}::\text{rend}(E) \) and \( \text{ranges}::\text{rbegin}(E) \) model \( \text{sentinel\_for}<S, I> \). — end note]

24.3.8 \( \text{ranges}::\text{crbegin} \) [range.access.crbegin]

The name \( \text{ranges}::\text{crbegin} \) denotes a customization point object (16.3.3.3.6). The expression \( \text{ranges}::\text{crbegin}(E) \) for a subexpression \( E \) of type \( T \) is expression-equivalent to:

\[
\begin{align*}
(1.1) & \quad \text{ranges}::\text{rbegin}(\text{static\_cast}<\text{const } T&>(E)) \text{ if } E \text{ is an lvalue.} \\
(1.2) & \quad \text{Otherwise, ranges}::\text{rbegin}(\text{static\_cast}<\text{const } T&&>(E)).
\end{align*}
\]

2 [Note 1: Whenever \( \text{ranges}::\text{crbegin}(E) \) is a valid expression, its type models \( \text{input\_or\_output\_iterator} \). — end note]

24.3.9 \( \text{ranges}::\text{crend} \) [range.access.crend]

The name \( \text{ranges}::\text{crend} \) denotes a customization point object (16.3.3.3.6). The expression \( \text{ranges}::\text{crend}(E) \) for a subexpression \( E \) of type \( T \) is expression-equivalent to:

\[
\begin{align*}
(1.1) & \quad \text{ranges}::\text{rend}(\text{static\_cast}<\text{const } T&>(E)) \text{ if } E \text{ is an lvalue.} \\
(1.2) & \quad \text{Otherwise, ranges}::\text{rend}(\text{static\_cast}<\text{const } T&&>(E)).
\end{align*}
\]

2 [Note 1: Whenever \( \text{ranges}::\text{crend}(E) \) is a valid expression, the types \( S \) and \( I \) of the expressions \( \text{ranges}::\text{rend}(E) \) and \( \text{ranges}::\text{crbegin}(E) \) model \( \text{sentinel\_for}<S, I> \). — end note]

24.3.10 \( \text{ranges}::\text{size} \) [range.prim.size]

The name \( \text{ranges}::\text{size} \) denotes a customization point object (16.3.3.3.6).

2 Given a subexpression \( E \) with type \( T \), let \( t \) be an lvalue that denotes the reified object for \( E \). Then:

\[
\begin{align*}
(2.1) & \quad \text{If } T \text{ is an array of unknown bound (9.3.4.5), ranges}::\text{size}(E) \text{ is ill-formed.} \\
(2.2) & \quad \text{Otherwise, if } T \text{ is an array type, ranges}::\text{size}(E) \text{ is expression-equivalent to } \text{decay\_copy}(\text{extent\_v}<T>). \\
(2.3) & \quad \text{Otherwise, if disable\_sized\_range<remove\_cv\_t<T>>(24.4.3) is false and } \text{decay\_copy}(t.\text{size}()) \text{ is a valid expression of integer-like type (23.3.4.4), ranges}::\text{size}(E) \text{ is expression-equivalent to } \text{decay\_copy}(t.\text{size}()). \text{\quad (2.4) Otherwise, if } T \text{ is a class or enumeration type, disable\_sized\_range<remove\_cv\_t<T>>(24.4.3) is false and } \text{decay\_copy}(\text{size}(t)) \text{ is a valid expression of integer-like type with overload resolution performed in a context in which unqualified lookup for } \text{size} \text{ finds only the declarations}
\end{align*}
\]

```cpp
void size(auto&) = delete;
void size(const auto&) = delete;
```
then \texttt{ranges::size(E)} is expression-equivalent to \texttt{decay-copy(size(t))} with overload resolution performed in the above context.

(2.5) — Otherwise, if \texttt{to-unsigned-like(ranges::end(t) - ranges::begin(t))} (24.2) is a valid expression and the types \(I\) and \(S\) of \texttt{ranges::begin(t)} and \texttt{ranges::end(t)} (respectively) model both \texttt{sized_sentinel_for\langle S, T\rangle} (23.3.4.8) and \texttt{forward_iterator\langle I\rangle}, then \texttt{ranges::size(E)} is expression-equivalent to \texttt{to-unsigned-like(ranges::end(t) - ranges::begin(t))}.

(2.6) — Otherwise, \texttt{ranges::size(E)} is ill-formed.

\[\text{Note 1: Diagnosable ill-formed cases above result in substitution failure when ranges::size(E) appears in the immediate context of a template instantiation.} \quad \text{—end note}\]

\[\text{Note 2: Whenever ranges::size(E) is a valid expression, its type is integer-like.} \quad \text{—end note}\]

24.3.11 \texttt{ranges::ssize} \hfill [range.prim.ssize]

1 The name \texttt{ranges::ssize} denotes a customization point object (16.3.3.3.6). The expression \texttt{ranges::ssize(E)} for a subexpression \(E\) of type \(T\) is expression-equivalent to:

(1.1) — If \texttt{range_difference_t\langle T\rangle} has width less than \texttt{ptrdiff_t}, \texttt{static_cast<ptrdiff_t>(ranges::size(E))}.

(1.2) — Otherwise, \texttt{static_cast<range_difference_t\langle T\rangle>(ranges::size(E))}.

24.3.12 \texttt{ranges::empty} \hfill [range.prim.empty]

1 The name \texttt{ranges::empty} denotes a customization point object (16.3.3.3.6).

2 Given a subexpression \(E\) with type \(T\), let \(t\) be an lvalue that denotes the reified object for \(E\). Then:

(2.1) — If \(T\) is an array of unknown bound (6.8.3), \texttt{ranges::empty(E)} is ill-formed.

(2.2) — Otherwise, if \texttt{bool(t.empty())} is a valid expression, \texttt{ranges::empty(E)} is expression-equivalent to \texttt{bool(t.empty())}.

(2.3) — Otherwise, if \texttt{(ranges::size(t) == 0)} is a valid expression, \texttt{ranges::empty(E)} is expression-equivalent to \texttt{(ranges::size(t) == 0)}.

(2.4) — Otherwise, if \texttt{bool(ranges::begin(t) == ranges::end(t))} is a valid expression and the type of \texttt{ranges::begin(t)} models \texttt{forward_iterator}, \texttt{ranges::empty(E)} is expression-equivalent to \texttt{bool(ranges::begin(t) == ranges::end(t))}.

(2.5) — Otherwise, \texttt{ranges::empty(E)} is ill-formed.

\[\text{Note 1: Diagnosable ill-formed cases above result in substitution failure when ranges::empty(E) appears in the immediate context of a template instantiation.} \quad \text{—end note}\]

\[\text{Note 2: Whenever ranges::empty(E) is a valid expression, it has type bool.} \quad \text{—end note}\]

24.3.13 \texttt{ranges::data} \hfill [range.prim.data]

1 The name \texttt{ranges::data} denotes a customization point object (16.3.3.3.6).

2 Given a subexpression \(E\) with type \(T\), let \(t\) be an lvalue that denotes the reified object for \(E\). Then:

(2.1) — If \(E\) is an rvalue and \texttt{enable_borrowed_range<remove_cv_t\langle T\rangle>} is false, \texttt{ranges::data(E)} is ill-formed.

(2.2) — Otherwise, if \(T\) is an array type (6.8.3) and \texttt{remove_all_extents_t\langle T\rangle} is an incomplete type, \texttt{ranges::data(E)} is ill-formed with no diagnostic required.

(2.3) — Otherwise, if \texttt{decay-copy(t.data())} is a valid expression of pointer to object type, \texttt{ranges::data(E)} is expression-equivalent to \texttt{decay-copy(t.data())}.

(2.4) — Otherwise, if \texttt{ranges::begin(t)} is a valid expression whose type models \texttt{contiguous_iterator}, \texttt{ranges::data(E)} is expression-equivalent to \texttt{to_address(ranges::begin(E))}.

(2.5) — Otherwise, \texttt{ranges::data(E)} is ill-formed.

\[\text{Note 1: Diagnosable ill-formed cases above result in substitution failure when ranges::data(E) appears in the immediate context of a template instantiation.} \quad \text{—end note}\]

\[\text{Note 2: Whenever ranges::data(E) is a valid expression, it has pointer to object type.} \quad \text{—end note}\]
24.3.14 ranges::cdata

The name ranges::cdata denotes a customization point object (16.3.3.3.6). The expression ranges::
cdata(E) for a subexpression E of type T is expression-equivalent to:

1. If E is an lvalue:
   - ranges::data(static_cast<const T&>(E))
2. Otherwise:
   - ranges::data(static_cast<const T&&>(E)).

[Note 1: Whenever ranges::cdata(E) is a valid expression, it has pointer to object type. —end note]

24.4 Range requirements

24.4.1 General

Ranges are an abstraction that allow a C++ program to operate on elements of data structures uniformly. Calling ranges::begin on a range returns an object whose type models input_or_output_iterator (23.3.4.6). Calling ranges::end on a range returns an object whose type S, together with the type I of the object returned by ranges::begin, models sentinel_for<S, I>. The library formalizes the interfaces, semantics, and complexity of ranges to enable algorithms and range adaptors that work efficiently on different types of sequences.

The range concept requires that ranges::begin and ranges::end return an iterator and a sentinel, respectively. The sized_range concept refines range with the requirement that ranges::size be amortized \(O(1)\). The view concept specifies requirements on a range type with constant-time destruction and move operations.

Several refinements of range group requirements that arise frequently in concepts and algorithms. Common ranges are ranges for which ranges::begin and ranges::end return objects of the same type. Random access ranges are ranges for which ranges::begin returns a type that models random_access_iterator (23.3.4.13). (Contiguous, bidirectional, forward, input, and output ranges are defined similarly.) Viewable ranges can be converted to views.

24.4.2 Ranges

The range concept defines the requirements of a type that allows iteration over its elements by providing an iterator and sentinel that denote the elements of the range.

```cpp
template<class T>
concept range =
    requires(T& t) {
        ranges::begin(t); // sometimes equality-preserving (see below)
        ranges::end(t);
    };
```

The required expressions ranges::begin(t) and ranges::end(t) of the range concept do not require implicit expression variations (18.2).

Given an expression t such that decltype((t)) is T, T models range only if

1. [ranges::begin(t), ranges::end(t)) denotes a range (23.3.1),
2. both ranges::begin(t) and ranges::end(t) are amortized constant time and non-modifying, and
3. if the type of ranges::begin(t) models forward_iterator, ranges::begin(t) is equality-preserving.

[Note 1: Equality preservation of both ranges::begin and ranges::end enables passing a range whose iterator type models forward_iterator to multiple algorithms and making multiple passes over the range by repeated calls to ranges::begin and ranges::end. Since ranges::begin is not required to be equality-preserving when the return type does not model forward_iterator, repeated calls might not return equal values or might not be well-defined. —end note]

```cpp
template<class T>
concept borrowed_range =
    range<T> &&
    (is_lvalue_reference_v<T> || enable_borrowed_range<remove_cvref_t<T>>);
```

Given an expression E such that decltype((E)) is T, T models borrowed_range only if the validity of iterators obtained from the object denoted by E is not tied to the lifetime of that object.
[Note 2: Since the validity of iterators is not tied to the lifetime of an object whose type models borrowed_range, a function can accept arguments of such a type by value and return iterators obtained from it without danger of dangling. — end note]

```cpp
template<class>
inline constexpr bool enable_borrowed_range = false;
```

Remarks: Pursuant to 16.4.5.2.1, users may specialize enable_borrowed_range for cv-unqualified program-defined types. Such specializations shall be usable in constant expressions (7.7) and have type const bool.

[Example 1: Each specialization $S$ of class template subrange (24.5.4) models borrowed_range because

1. $S$'s iterators do not have validity tied to the lifetime of an $S$ object because they are “borrowed” from some other range.

— end example]

24.4.3 Sized ranges

The sized_range concept refines range with the requirement that the number of elements in the range can be determined in amortized constant time using ranges::size.

```cpp
template<class T>
concept sized_range =
    range<T> &&
    requires(T& t) { ranges::size(t); };
```

Given an lvalue $t$ of type remove_reference_t<T>, $T$ models sized_range only if

1. ranges::size(t) is amortized $\Theta(1)$, does not modify $t$, and is equal to ranges::distance(t), and
2. if iterator_t<T> models forward_iterator, ranges::size(t) is well-defined regardless of the evaluation of ranges::begin(t).

[Note 1: ranges::size(t) is otherwise not required to be well-defined after evaluating ranges::begin(t). For example, ranges::size(t) might be well-defined for a sized_range whose iterator type does not model forward_iterator only if evaluated before the first call to ranges::begin(t). — end note]

```cpp
template<class>
inline constexpr bool disable_sized_range = false;
```

Remarks: Pursuant to 16.4.5.2.1, users may specialize disable_sized_range for cv-unqualified program-defined types. Such specializations shall be usable in constant expressions (7.7) and have type const bool.

[Note 2: disable_sized_range allows use of range types with the library that satisfy but do not in fact model sized_range. — end note]

24.4.4 Views

The view concept specifies the requirements of a range type that has constant time move construction, move assignment, and destruction; that is, the cost of these operations is independent of the number of elements in the view.

```cpp
template<class T>
concept view =
    range<T> && movable<T> && default_initializable<T> && enable_view<T>;
```

$T$ models view only if:

1. $T$ has $\Theta(1)$ move construction; and
2. $T$ has $\Theta(1)$ move assignment; and
3. $T$ has $\Theta(1)$ destruction; and
4. copy_constructible<T> is false, or $T$ has $\Theta(1)$ copy construction; and
5. copyable<T> is false, or $T$ has $\Theta(1)$ copy assignment.
Example 1: Examples of views are:

— A range type that wraps a pair of iterators.
— A range type that holds its elements by shared_ptr and shares ownership with all its copies.
— A range type that generates its elements on demand.

Most containers (Clause 22) are not views since destruction of the container destroys the elements, which cannot be done in constant time.

Since the difference between range and view is largely semantic, the two are differentiated with the help of enable_view.

\[
\text{template<class T>}
\]
\[
\text{inline constexpr bool enable_view = derived_from<T, view_base>;}\]

Remarks: Pursuant to 16.4.5.2.1, users may specialize enable_view to true for cv-unqualified program-defined types which model view, and false for types which do not. Such specializations shall be usable in constant expressions (7.7) and have type const bool.

24.4.5 Other range refinements

The output_range concept specifies requirements of a range type for which ranges::begin returns a model of output_iterator (23.3.4.10). input_range, forward_range, bidirectional_range, and random_access_range are defined similarly.

\[
\text{template<class R, class T>}
\]
\[
\text{concept output_range = range<R> && output_iterator<iterator_t<R>, T>;}\]

\[
\text{template<class T>}
\]
\[
\text{concept input_range = range<T> && input_iterator<iterator_t<T>;}\]

\[
\text{template<class T>}
\]
\[
\text{concept forward_range = input_range<T> && forward_iterator<iterator_t<T>;}\]

\[
\text{template<class T>}
\]
\[
\text{concept bidirectional_range = forward_range<T> && bidirectional_iterator<iterator_t<T>;}\]

\[
\text{template<class T>}
\]
\[
\text{concept random_access_range = bidirectional_range<T> && random_access_iterator<iterator_t<T>;}\]

contiguous_range additionally requires that the ranges::data customization point object (24.3.13) is usable with the range.

\[
\text{template<class T>}
\]
\[
\text{concept contiguous_range = random_access_range<T> && contiguous_iterator<iterator_t<T>;}\]

Given an expression t such that decltype((t)) is T& T models contiguous_range only if to_address(ranges::begin(t)) == ranges::data(t) is true.

The common_range concept specifies requirements of a range type for which ranges::begin and ranges::end return objects of the same type.

Example 1: The standard containers (Clause 22) model common_range. — end example

\[
\text{template<class T>}
\]
\[
\text{concept common_range = range<T> && same_as<iterator_t<T>, sentinel_t<T>;}\]

The viewable_range concept specifies the requirements of a range type that can be converted to a view safely.
template<class T>
concept viewable_range =
    range<T> && (borrowed_range<T> || view<remove_cvref_t<T>>);

24.5 Range utilities

24.5.1 General

The components in 24.5 are general utilities for representing and manipulating ranges.

24.5.2 Helper concepts

Many of the types in subclause 24.5 are specified in terms of the following exposition-only concepts:

template<class R>
concept simple-view = // exposition only
    view<R> && range<const R> &&
    same_as<iterator_t<R>, iterator_t<const R>> &&
    same_as<sentinel_t<R>, sentinel_t<const R>>;

template<class I>
concept has-arrow = // exposition only
    input_iterator<I> && (is_pointer_v<I> || requires(I i) { i.operator->(); });

template<class T, class U>
concept not-same-as = // exposition only
    !same_as<remove_cvref_t<T>, remove_cvref_t<U>>;

24.5.3 View interface

24.5.3.1 General

The class template view_interface is a helper for defining view-like types that offer a container-like interface. It is parameterized with the type that is derived from it.

namespace std::ranges {
    template<class D>
    requires is_class_v<D> && same_as<D, remove_cv_t<D>>
    class view_interface : public view_base {
        private:
            constexpr D& derived() noexcept {
                // exposition only
                return static_cast<D&>(*this);
            }
            constexpr const D& derived() const noexcept {
                // exposition only
                return static_cast<const D&>(*this);
            }
            public:
                constexpr bool empty() requires forward_range<D> {
                    return ranges::begin(derived()) == ranges::end(derived());
                }
                constexpr bool empty() const requires forward_range<const D> {
                    return ranges::begin(derived()) == ranges::end(derived());
                }

                constexpr explicit operator bool() requires requires { ranges::empty(derived()); } {
                    return !ranges::empty(derived());
                }
                constexpr explicit operator bool() const requires requires { ranges::empty(derived()); } {
                    return !ranges::empty(derived());
                }

                constexpr auto data() requires contiguous_iterator<iterator_t<D>> {
                    return to_address(ranges::begin(derived()));
                }
    }
The template parameter \( D \) for `view_interface` may be an incomplete type. Before any member of the resulting specialization of `view_interface` other than special member functions is referenced, \( D \) shall be complete, and model both `derived_from<view_interface<D>>` and `view`.

### 24.5.3.2 Members

Preconditions: `!empty()`.

Effects: Equivalent to: `return *ranges::begin(derived());`

### 24.5.4 Sub-ranges

The `subrange` class template combines together an iterator and a sentinel into a single object that models the `view` concept. Additionally, it models the `sized_range` concept when the final template parameter is `subrange_kind::sized`.

```cpp
namespace std::ranges {
    template<class From, class To>
    concept convertible_to_non_slicing = // exposition only
        convertible_to<From, To> &&
        !(is_pointer_v<decay_t<From>> &&
         is_pointer_v<decay_t<To>> &&
         not_same_as<remove_pointer_t<decay_t<From>>, remove_pointer_t<decay_t<To>>>);
```
template<class T>
concept pair-like = // exposition only

!is_reference_v<T> && requires(T t) {
  typename tuple_size<T>::type; // ensures tuple_size<T> is complete
  requires derived_from<tuple_size<T>, integral_constant<size_t, 2>>;
  typename tuple_element_t<0, remove_const_t<T>>;
  typename tuple_element_t<1, remove_const_t<T>>;
  {get<0>(t)} -> convertible_to<const tuple_element_t<0, T>&>;
  {get<1>(t)} -> convertible_to<const tuple_element_t<1, T>&>;
};

template<class T, class U, class V>
concept pair-like-convertible-from = // exposition only

!range<T> && pair-like<T> &&
constructible_from<T, U, V> &&
convertible-to-non-slicing<U, tuple_element_t<0, T>>, T>> &&
convertible_to<V, tuple_element_t<1, T>>;

template<class T>
concept iterator-sentinel-pair = // exposition only

!range<T> && pair-like<T> &&
sentinel_for<tuple_element_t<1, T>, tuple_element_t<0, T>>;

template<input_or_output_iterator I, sentinel_for<I> S = I, subrange_kind K =
sized_sentinel_for<S, I> ? subrange_kind::sized : subrange_kind::unsized>
requires (K == subrange_kind::sized || !sized_sentinel_for<S, I>)
class subrange : public view_interface<subrange<I, S, K>> {
private:
  static constexpr bool StoreSize = // exposition only
    K == subrange_kind::sized && !sized_sentinel_for<S, I>;
  I begin_ = I(); // exposition only
  S end_ = S(); // exposition only
  make_unsigned_like_t<iter_difference_t<I>> size_ = 0; // exposition only; present only
  // when StoreSize is true
public:
  subrange() = default;

constexpr subrange(convertible-to-non-slicing<I> auto i, S s) requires (!StoreSize);

constexpr subrange(convertible-to-non-slicing<I> auto i, S s,
  make_unsigned_like_t<iter_difference_t<I>> n) requires (K == subrange_kind::sized);

template<not_same_as<subrange> R>
requires borrowed_range<R> &&
  convertible-to-non-slicing<iterator_t<R>, I> &&
  convertible_to<sentinel_t<R>, S>
constexpr subrange(R&& r) requires (!StoreSize || sized_range<R>);

template<borrowed_range R>
requires convertible-to-non-slicing<iterator_t<R>, I> &&
  convertible_to<sentinel_t<R>, S>
constexpr subrange(R&& r, make_unsigned_like_t<iter_difference_t<I>> n) requires (K == subrange_kind::sized)
  : subrange(ranges::begin(r), ranges::end(r), n) {}

template<not_same_as<subrange> PairLike>
requires pair-like-convertible-from<PairLike, const I&, const S&>
constexpr operator PairLike() const;

constexpr I begin() const requires copyable<I>;
[[nodiscard]] constexpr I begin() requires (!copyable<I>);
constexpr S end() const;
constexpr bool empty() const;
constexpr make_unsigned-like-t<iter_difference_t<I>> size() const
requires (K == subrange_kind::sized);

[[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const &
requires forward_iterator<I>;

[[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) &&;

[[nodiscard]] constexpr subrange prev(iter_difference_t<I> n = 1) const
requires bidirectional_iterator<I>;

constexpr subrange& advance(iter_difference_t<I> n);

};

template<input_or_output_iterator I, sentinel_for<I> S>
subrange(I, S) -> subrange<I, S>;

template<input_or_output_iterator I, sentinel_for<I> S>
subrange(I, S,
make_unsigned-like-t<iter_difference_t<I>>) ->
subrange<I, S, subrange_kind::sized>;

template<iterator-sentinel-pair P>
subrange(P) -> subrange<tuple_element_t<0, P>, tuple_element_t<1, P>>;

template<iterator-sentinel-pair P>
subrange(P, make_unsigned-like-t<iter_difference_t<tuple_element_t<0, P>>>) ->
subrange<tuple_element_t<0, P>, tuple_element_t<1, P>, subrange_kind::sized>;

template<borrowed_range R>
subrange(R&&) ->
subrange<iterator_t<R>, sentinel_t<R>,
(sized_range<R> || sized_sentinel_for<sentinel_t<R>, iterator_t<R>>) ? subrange_kind::sized : subrange_kind::unsized>;

template<borrowed_range R>
subrange(R&&, make_unsigned-like-t<range_difference_t<R>>) ->
subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

template<size_t N, class I, class S, subrange_kind K>
requires (N < 2)
constexpr auto get(const subrange<I, S, K>& r);

};

namespace std {
using ranges::get;
}

24.5.4.2 Constructors and conversions [range.subrange.ctor]

constexpr subrange(convertible-to-non-slicing<I> auto i, S s) requires (!StoreSize);

1 Preconditions: [i, s) is a valid range.
2 Effects: Initializes begin_ with std::move(i) and end_ with s.

constexpr subrange(convertible-to-non-slicing<I> auto i, S s,
make_unsigned-like-t<iter_difference_t<I>> n)
requires (K == subrange_kind::sized);

3 Preconditions: [i, s) is a valid range, and n == to_unsigned-like(ranges::distance(i, s)).
4 Effects: Initializes begin_ with std::move(i) and end_ with s. If StoreSize is true, initializes size_ with n.
[Note 1: Accepting the length of the range and storing it to later return from \texttt{size()} enables \texttt{subrange} to model \texttt{sized\_range} even when it stores an iterator and sentinel that do not model \texttt{sized\_sentinel\_for}. — end note]

template<not-same-as<subrange> R>
requires borrowed_range<R> &&
convertible-to-non-slicing<

\texttt{iterator\_t<\texttt{R}>}, I> &&
convertible_to<

\texttt{sentinel\_t<\texttt{R}>}, S>

constexpr subrange(R&& r) requires (!StoreSize || sized_range<R>);

\textbf{Effects:} Equivalent to:

- If \texttt{StoreSize} is true, subrange\{r, ranges::size(r)\}.  
- Otherwise, subrange\{ranges::begin(r), ranges::end(r)\}.

template<not-same-as<subrange> PairLike>
requires pair-like-convertible-from<PairLike, const I&, const S&>

constexpr operator PairLike() const;

\textbf{Effects:} Equivalent to: return PairLike(begin_, end_);

\subsection*{24.5.4.3 Accessors} \hfill [range.subrange.access]

constexpr I begin() const requires copyable<I>;

\textbf{Effects:} Equivalent to: return begin_; 

[[nondiscard]] constexpr I begin() requires (!copyable<I>);  

\textbf{Effects:} Equivalent to: return std::move(begin_); 

constexpr S end() const;

\textbf{Effects:} Equivalent to: return end_;  

constexpr bool empty() const;

\textbf{Effects:} Equivalent to: return begin_ == end_; 

constexpr\

\texttt{make-unsigned\_like\_t<}\texttt{iter\_difference\_t<I>}> size() const  

requires (K == subrange\_kind::sized);

\textbf{Effects:}

- If \texttt{StoreSize} is true, equivalent to: return size_;  
- Otherwise, equivalent to: return \texttt{to\_unsigned\_like}(end_ - begin_);  

[[nondiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const &
requires forward_iterator<I>;

\textbf{Effects:} Equivalent to:

auto tmp = *this; 

\texttt{tmp.advance(n)}; 

return tmp;

[[nondiscard]] constexpr subrange next(iter_difference_t<I> n = 1) &&;

\textbf{Effects:} Equivalent to: advance(n); 

return std::move(*this);

[[nondiscard]] constexpr subrange prev(iter_difference_t<I> n = 1) const
requires bidirectional_iterator<I>;

\textbf{Effects:} Equivalent to:

auto tmp = *this; 

\texttt{tmp.advance(-n)}; 

return tmp;

constexpr subrange& advance(iter_difference_t<I> n);

\textbf{Effects:} Equivalent to:
If `StoreSize` is true,

```cpp
    auto d = n - ranges::advance(begin_, n, end_);
    if (d >= 0)
        size_ -= to_unsigned_like(d);
    else
        size_ += to_unsigned_like(-d);
    return *this;
```

Otherwise,

```cpp
    ranges::advance(begin_, n, end_);
    return *this;
```

**Template**

```cpp
template<size_t N, class I, class S, subrange_kind K>
requires (N < 2)
constexpr auto get(const subrange<I, S, K>& r);
```

```cpp
template<size_t N, class I, class S, subrange_kind K>
requires (N < 2)
constexpr auto get(subrange<I, S, K>&& r);
```

**Effects:** Equivalent to:

```cpp
    if constexpr (N == 0)
        return r.begin();
    else
        return r.end();
```

### 24.5.5 Dangling iterator handling

**Example 1:**

```cpp
vector<int> f();
auto result1 = ranges::find(f(), 42); // #1
static_assert(same_as<decltype(result1), ranges::dangling>);
auto vec = f();
auto result2 = ranges::find(vec, 42); // #2
static_assert(same_as<decltype(result2), vector<int>::iterator>);
auto result3 = ranges::find(subrange{vec}, 42); // #3
static_assert(same_as<decltype(result3), vector<int>::iterator>);
```

The call to `ranges::find` at #1 returns `ranges::dangling` since `f()` is an rvalue `vector`; the `vector` could potentially be destroyed before a returned iterator is dereferenced. However, the calls at #2 and #3 both return iterators since the lvalue `vec` and specializations of `subrange` model `borrowed_range`.

---

**Range factories**

**General**

Subclause 24.6 defines `range factories`, which are utilities to create a `view`.

Range factories are declared in namespace `std::ranges::views`.

**Empty view**

**Overview**

`empty_view` produces a `view` of no elements of a particular type.
empty_view<int> e;
static_assert(ranges::empty(e));
static_assert(0 == e.size());
— end example]

24.6.2.2 Class template empty_view

namespace std::ranges {
    template<class T>
    requires is_object_v<T>
    class empty_view : public view_interface<empty_view<T>> {
        public:
            static constexpr T* begin() noexcept { return nullptr; }
            static constexpr T* end() noexcept { return nullptr; }
            static constexpr T* data() noexcept { return nullptr; }
            static constexpr size_t size() noexcept { return 0; }
            static constexpr bool empty() noexcept { return true; }
    };
}

24.6.3 Single view

24.6.3.1 Overview

single_view produces a view that contains exactly one element of a specified value.

The name views::single denotes a customization point object (16.3.3.6). Given a subexpression E, the expression views::single(E) is expression-equivalent to single_view{E}.

[Example 1]:

    single_view s{4};
    for (int i : s)
        cout << i;  // prints 4
— end example]

24.6.3.2 Class template single_view

namespace std::ranges {
    template<copy_constructible T>
    requires is_object_v<T>
    class single_view : public view_interface<single_view<T>> {
        private:
            semiregular-box<T> value_;  // exposition only (see 24.7.3)
        public:
            single_view() = default;
            constexpr explicit single_view(const T& t);
            constexpr explicit single_view(T&& t);
            template<class... Args>
                requires constructible_from<T, Args...>
                constexpr single_view(in_place_t, Args&&... args);
            constexpr T* begin() noexcept;
            constexpr const T* begin() const noexcept;
            constexpr T* end() noexcept;
            constexpr const T* end() const noexcept;
            static constexpr size_t size() noexcept;
            constexpr T* data() noexcept;
            constexpr const T* data() const noexcept;
    };

    constexpr explicit single_view(const T& t);
    1 Effects: Initializes value_ with t.

    constexpr explicit single_view(T&& t);
    2 Effects: Initializes value_ with std::move(t).

§ 24.6.3.2 999
template<class... Args>
  requires constructible_from<T, Args...>
  constexpr single_view(in_place_t, Args&&... args);

Effects: Initializes value_ as if by value_{in_place, std::forward<Args>(args)...}.

constexpr T* begin() noexcept;
constexpr const T* begin() const noexcept;

Effects: Equivalent to: return data();

constexpr T* end() noexcept;
constexpr const T* end() const noexcept;

Effects: Equivalent to: return data() + 1;

static constexpr size_t size() noexcept;

constexpr T* data() noexcept;
constexpr const T* data() const noexcept;

Effects: Equivalent to: return value_.operator->();

§ 24.6.4  Iota view  [range.iota]

24.6.4.1  Overview  [range.iota.overview]

iota_view generates a sequence of elements by repeatedly incrementing an initial value.

The name views::iota denotes a customization point object (16.3.3.3.6). Given subexpressions E and F, the expressions views::iota(E) and views::iota(E, F) are expression-equivalent to iota_view{E} and iota_view{E, F}, respectively.

[Example 1:]

for (int i : iota_view{1, 10})
  cout << i << ' '; // prints: 1 2 3 4 5 6 7 8 9
— end example]

24.6.4.2  Class template iota_view  [range.iota.view]

namespace std::ranges {
  template<class I>
    concept decrementable = // exposition only
      see below;
  template<class I>
    concept advanceable = // exposition only
      see below;

  template<weakly_incrementable W, semiregular Bound = unreachable_sentinel_t>
    requires weakly-equality-comparable-with<W, Bound> && semiregular<W>
  class iota_view : public view_interface<iota_view<W, Bound>> {
    private:
      // 24.6.4.3, class iota_view::iterator
      struct iterator; // exposition only
      // 24.6.4.4, class iota_view::sentinel
      struct sentinel; // exposition only

      W value_ = W(); // exposition only
      Bound bound_ = Bound(); // exposition only

    public:
      iota_view() = default;
      constexpr explicit iota_view(W value);
      constexpr iota_view(type_identity_t<W> value,
        type_identity_t<Bound> bound);
      constexpr iota_view(iterator first, sentinel last) : iota_view(*first, last.bound_) {}

      constexpr iterator begin() const;
      constexpr auto end() const;
      constexpr iterator end() const requires same_as<W, Bound>;
constexpr auto size() const requires see below;

```cpp
template<class W, class Bound>
requires (!is-integer-like<W> || !is-integer-like<Bound> ||
    (is-signed-integer-like<W> == is-signed-integer-like<Bound>))
iota_view(W, Bound) -> iota_view<W, Bound>;
```

1 Let \( IOTA-DIFF-T(W) \) be defined as follows:

1.1 — If \( W \) is not an integral type, or if it is an integral type and sizeof(iter_difference_t\(<W>\)) is greater than sizeof(\( W \)), then \( IOTA-DIFF-T(W) \) denotes iter_difference_t\(<W>\).

1.2 — Otherwise, \( IOTA-DIFF-T(W) \) is a signed integer type of width greater than the width of \( W \) if such a type exists.

1.3 — Otherwise, \( IOTA-DIFF-T(W) \) is an unspecified signed-integer-like type (23.3.4.4) of width not less than the width of \( W \).

[Note 1: It is unspecified whether this type satisfies weakly_incrementable. — end note]

2 The exposition-only decrementable concept is equivalent to:

```cpp
template<class I>
concept decrementable =
    incremental<1> && requires(I i) {
    { --i } -> same_as<I&>;
    { i-- } -> same_as<I>;
    };
```

When an object is in the domain of both pre- and post-decrement, the object is said to be decrementable.

3 Let a and b be equal objects of type I. I models decrementable only if

4.1 — If a and b are decrementable, then the following are all true:

4.1.1 — addressof(--a) == addressof(a)

4.1.2 — bool(a-- == b)

4.1.3 — bool(((void)a--, a) == --b)

4.1.4 — bool(++(--a) == b).

4.2 — If a and b are incrementable, then bool(--(++a) == b).

5 The exposition-only advanceable concept is equivalent to:

```cpp
template<class I>
concept advanceable =
    decrementable<I> && totally_ordered<I> &&
    requires(I i, const I j, const IOTA-DIFF-T(I) n) {
    { i += n } -> same_as<I&>;
    { i -= n } -> same_as<I&>;
    I(j + n);
    I(n + j);
    I(j - n);
    { j - j } -> convertible_to<IOTA-DIFF-T(I)>;
    };
```

Let D be \( IOTA-DIFF-T(I) \). Let a and b be objects of type I such that b is reachable from a after n applications of ++a, for some value n of type D. I models advanceable only if

5.1 — (a += n) is equal to b.

5.2 — addressof(a += n) is equal to addressof(a).

5.3 — I(a + n) is equal to (a += n).

5.4 — For any two positive values x and y of type D, if I(a + D(x + y)) is well-defined, then I(a + D(x + y)) is equal to I(I(a + x) + y).

5.5 — I(a + D(0)) is equal to a.
If \( I(a + D(n - 1)) \) is well-defined, then \( I(a + n) \) is equal to \( (I(c) \{ return ++c; \})(I(a + D(n - 1))) \).

(b += -n) is equal to a.

(b -= n) is equal to a.

\text{addressof}(b -= n) \text{ is equal to } \text{addressof}(b).

I(b - n) \text{ is equal to } (b -= n).

D(b - a) \text{ is equal to } D(-n).

\text{bool}(a <= b) \text{ is true.}

\text{constexpr explicit iota_view}(W \text{ value});

\text{Preconditions:} Bound \text{ denotes unreachable_sentinel_t or } Bound() \text{ is reachable from value.}

\text{Effects:} \text{Initializes value_ with value.}

\text{constexpr iota_view}(\text{type_identity_t}<W> \text{ value, type_identity_t}<\text{Bound}> \text{ bound});

\text{Preconditions:} Bound \text{ denotes unreachable_sentinel_t or } \text{bound is reachable from value. When } W \text{ and Bound model totally_ordered_with, then bool(value <= bound) is true.}

\text{Effects:} \text{Initializes value_ with value and bound_ with bound.}

\text{constexpr iterator begin()} \text{ const;}

\text{Effects:} \text{Equivalent to: return iterator(value_);} 

\text{constexpr auto end()} \text{ const;}

\text{Effects:} \text{Equivalent to:}

\begin{verbatim}
if constexpr (same_as<Bound, unreachable_sentinel_t>)
    return unreachable_sentinel;
else
    return sentinel{bound_};
\end{verbatim}

\text{constexpr iterator end()} \text{ const requires same_as<W, Bound>;}

\text{Effects:} \text{Equivalent to: return iterator(bound_);} 

\text{constexpr auto size()} \text{ const requires see below;}

\text{Effects:} \text{Equivalent to:}

\begin{verbatim}
if constexpr (is-integer-like<W> \&\& is-integer-like<Bound>)
    return (value_ < 0)
        ? ((bound_ < 0)
            ? to-unsigned-like(-value_) - to-unsigned-like(-bound_)
            : to-unsigned-like(bound_) + to-unsigned-like(-value_))
            : to-unsigned-like(bound_) - to-unsigned-like(value_);
    else 
        return to-unsigned-like(bound_ - value_);
\end{verbatim}

\text{Remarks:} \text{The expression in the requires-clause is equivalent to:}
\begin{verbatim}
(same_as<W, Bound> \&\& advanceable<W>) || (integral<W> \&\& integral<Bound>) || sized_sentinel_for<Bound, W>
\end{verbatim}

\textbf{24.6.4.3} Class iota_view::iterator

\begin{verbatim}
namespace std::ranges {
    template<weakly_incrementable W, semiregular Bound>
    requires weakly-equality-comparable-with<W, Bound>
    struct iota_view<W, Bound>::iterator {
        private:
            W value_ = W(); // exposition only
        public:
            using iterator_concept = see below;
            using iterator_category = input_iterator_tag;

    };
}
\end{verbatim}
using value_type = W;
using difference_type = IOTA-DIFF-T(W);

iterator() = default;
constexpr explicit iterator(W value);

constexpr W operator*() const noexcept(is_nothrow_copy_constructible_v<W>);
constexpr iterator& operator++();
constexpr void operator++(int);
constexpr iterator operator++(int) requires incrementable<W>;
constexpr iterator& operator--() requires decrementable<W>;
constexpr iterator operator--(int) requires decrementable<W>;
constexpr iterator& operator+=(difference_type n) requires advanceable<W>;
constexpr iterator& operator-=(difference_type n) requires advanceable<W>;
constexpr W operator[](difference_type n) const requires advanceable<W>;

friend constexpr bool operator==(const iterator& x, const iterator& y) requires equality_comparable<W>;
friend constexpr bool operator<(const iterator& x, const iterator& y) requires totally_ordered<W>;
friend constexpr bool operator>(const iterator& x, const iterator& y) requires totally_ordered<W>;
friend constexpr bool operator<=(const iterator& x, const iterator& y) requires totally_ordered<W>;
friend constexpr bool operator>=(const iterator& x, const iterator& y) requires totally_ordered<W>;
friend constexpr auto operator<=>(const iterator& x, const iterator& y) requires totally_ordered<W> && three_way_comparable<W>;
friend constexpr iterator operator+(iterator i, difference_type n) requires advanceable<W>;
friend constexpr iterator operator+(difference_type n, iterator i) requires advanceable<W>;
friend constexpr iterator operator-(iterator i, difference_type n) requires advanceable<W>;
friend constexpr difference_type operator-(const iterator& x, const iterator& y) requires advanceable<W>;

1 iterator::iterator_concept is defined as follows:
(1.1) — If W models advanceable, then iterator_concept is random_access_iterator_tag.
(1.2) — Otherwise, if W models decrementable, then iterator_concept is bidirectional_iterator_tag.
(1.3) — Otherwise, if W models incrementable, then iterator_concept is forward_iterator_tag.
(1.4) — Otherwise, iterator_concept is input_iterator_tag.

2 [Note 1: Overloads for iter_move and iter_swap are omitted intentionally. — end note]

constexpr explicit iterator(W value);

 Effects: Initializes value_ with value.

constexpr W operator*() const noexcept(is_nothrow_copy_constructible_v<W>);

 Effects: Equivalent to: return value_;

3

4 [Note 2: The noexcept clause is needed by the default iter_move implementation. — end note]
constexpr iterator operator++();

Effects: Equivalent to:

++value_;  
return *this;

constexpr void operator++(int);

Effects: Equivalent to +++this.

constexpr iterator operator++(int) requires incrementable<W>;

Effects: Equivalent to:

auto tmp = *this;
++*this;
return tmp;

constexpr iterator operator--() requires decrementable<W>;

Effects: Equivalent to:

--value_;  
return *this;

constexpr iterator operator--(int) requires decrementable<W>;

Effects: Equivalent to:

auto tmp = *this;
--*this;
return tmp;

constexpr iterator operator+=(difference_type n) requires advanceable<W>;

Effects: Equivalent to:

if constexpr (is-integer-like<W> && !is-signed-integer-like<W>) {
  if (n >= difference_type(0))
    value_ += static_cast<W>(n);
  else
    value_ -= static_cast<W>(-n);
} else {
  value_ += n;
}
return *this;

constexpr iterator operator-=(difference_type n) requires advanceable<W>;

Effects: Equivalent to:

if constexpr (is-integer-like<W> && !is-signed-integer-like<W>) {
  if (n >= difference_type(0))
    value_ -= static_cast<W>(n);
  else
    value_ += static_cast<W>(-n);
} else {
  value_ -= n;
}
return *this;

constexpr W operator[](difference_type n) const
requires advanceable<W>;

Effects: Equivalent to: return W(value_ + n);

friend constexpr bool operator==(const iterator& x, const iterator& y)
requires equality_comparable<W>;

Effects: Equivalent to: return x.value_ == y.value_;

§ 24.6.4.3
friend constexpr bool operator<(const iterator& x, const iterator& y) requires totally_ordered<W>;

   Effects: Equivalent to: return x.value_ < y.value_;

friend constexpr bool operator>(const iterator& x, const iterator& y) requires totally_ordered<W>;

   Effects: Equivalent to: return y < x;

friend constexpr bool operator<=(const iterator& x, const iterator& y) requires totally_ordered<W>;

   Effects: Equivalent to: return !(y < x);

friend constexpr bool operator>=(const iterator& x, const iterator& y) requires totally_ordered<W>;

   Effects: Equivalent to: return !(x < y);

friend constexpr auto operator<=>(const iterator& x, const iterator& y) requires totally_ordered<W> && three_way_comparable<W>;

   Effects: Equivalent to: return x.value_ <=> y.value_;

friend constexpr iterator operator+(iterator i, difference_type n) requires advanceable<W>;

   Effects: Equivalent to: return i += n;

friend constexpr iterator operator+(difference_type n, iterator i) requires advanceable<W>;

   Effects: Equivalent to: return i + n;

friend constexpr iterator operator-(iterator i, difference_type n) requires advanceable<W>;

   Effects: Equivalent to: return i -= n;

friend constexpr difference_type operator-(const iterator& x, const iterator& y) requires advanceable<W>;

   Effects: Equivalent to:
   using D = difference_type;
   if constexpr (is-integer-like<W>) {
      if constexpr (is-signed-integer-like<W>)
         return D(D(x.value_) - D(y.value_));
      else
         return (y.value_ > x.value_)
            ? D(-D(y.value_ - x.value_))
            : D(x.value_ - y.value_);
   } else {
      return x.value_ - y.value_;
   }

24.6.4.4 Class iota_view::sentinel

namespace std::ranges {  
   template<weakly_incrementable W, semiregular Bound>
      requires weakly-equality-comparable-with<W, Bound>
   struct iota_view<W, Bound>::sentinel {
      private:
         Bound bound_ = Bound();    // exposition only
      public:
         sentinel() = default;
         constexpr explicit sentinel(Bound bound);

         friend constexpr bool operator==(const iterator& x, const sentinel& y);

§ 24.6.4.4 1005
friend constexpr iter_difference_t<W> operator-(const iterator& x, const sentinel& y)
  requires sized_sentinel_for<Bound, W>;
friend constexpr iter_difference_t<W> operator-(const sentinel& x, const iterator& y)
  requires sized_sentinel_for<Bound, W>;
};
}
constexpr explicit sentinel(Bound bound);

Effects: Initializes bound_ with bound.

friend constexpr bool operator==(const iterator& x, const sentinel& y);

Effects: Equivalent to: return x.value_ == y.bound_

friend constexpr iter_difference_t<W> operator-(const iterator& x, const sentinel& y)
  requires sized_sentinel_for<Bound, W>;

Effects: Equivalent to: return x.value_ - y.bound_

friend constexpr iter_difference_t<W> operator-(const sentinel& x, const iterator& y)
  requires sized_sentinel_for<Bound, W>;

Effects: Equivalent to: return -(y - x);

24.6.5 Istream view

24.6.5.1 Overview

basic_istream_view models input_range and reads (using operator>>) successive elements from its corresponding input stream.

Example 1:

auto ints = istringstream("0 1 2 3 4");
ranges::copy(ranges::istream_view<int>(ints), ostream_iterator<int>{cout, "-"});
// prints 0-1-2-3-4-
— end example]

24.6.5.2 Class template basic_istream_view

namespace std::ranges {
  template<class Val, class CharT, class Traits>
  concept stream-extractable =
    requires(basic_istream<CharT, Traits>& is, Val& t) {
      is >> t;
    };

  template<movable Val, class CharT, class Traits>
  requires default_initializable<Val> &&
    stream-extractable<Val, CharT, Traits>
  class basic_istream_view : public view_interface<basic_istream_view<Val, CharT, Traits>> {
    public:
      basic_istream_view() = default;
      constexpr explicit basic_istream_view(basic_istream<CharT, Traits>&& stream);

      constexpr auto begin()
      {
        if (stream_) {
          *stream_ >> object_;
        }
        return iterator{*this};
      }

      constexpr default_sentinel_t end() const noexcept;

    private:
      struct iterator;
      basic_istream<CharT, Traits>** stream_ = nullptr;
      Val object_ = Val();
  };
}
constexpr explicit basic_istream_view(basic_istream<CharT, Traits>& stream);

Effects: Initializes stream with addressof(stream).

constexpr default_sentinel_t end() const noexcept;

Effects: Equivalent to: return default_sentinel;

template<class Val, class CharT, class Traits>
basic_istream_view<Val, CharT, Traits> istream_view(basic_istream<CharT, Traits>& s);

Effects: Equivalent to: return basic_istream_view<Val, CharT, Traits>{s};

24.6.5.3 Class template basic_istream_view::iterator

namespace std::ranges {

template<movable Val, class CharT, class Traits>
requires default_initializable<Val> &&
stream-extractable<Val, CharT, Traits>
class basic_istream_view<Val, CharT, Traits>::iterator {
  using iterator_concept = input_iterator_tag;
  using difference_type = ptrdiff_t;
  using value_type = Val;

  iterator() = default;
  constexpr explicit iterator(basic_istream_view& parent) noexcept;

  iterator(const iterator&) = delete;
  iterator(iterator&&) = default;

  iterator& operator=(const iterator&);
  iterator& operator=(iterator&&);

  iterator& operator++();
  void operator++(int);

  Val& operator*() const;

  friend bool operator==(const iterator& x, default_sentinel_t);

private:
  basic_istream_view* parent_ = nullptr;
}

constexpr explicit iterator(basic_istream_view& parent) noexcept;

Effects: Initializes parent with addressof(parent).

iterator& operator++();

Preconditions: parent->stream_ != nullptr is true.

Effects: Equivalent to:
*parent->stream_ >> parent->object_.
return *this;

void operator++(int);

Preconditions: parent->stream_ != nullptr is true.

Effects: Equivalent to ++*this.

Val& operator*() const;

Preconditions: parent->stream_ != nullptr is true.
24.7 Range adaptors

24.7.1 General

Subclause 24.7 defines range adaptors, which are utilities that transform a range into a view with custom behaviors. These adaptors can be chained to create pipelines of range transformations that evaluate lazily as the resulting view is iterated.

Range adaptors are declared in namespace `std::ranges::views`.

The bitwise OR operator is overloaded for the purpose of creating adaptor chain pipelines. The adaptors also support function call syntax with equivalent semantics.

[Example 1:

```cpp
#include <vector>
#include <ranges>

std::vector<int> ints{0,1,2,3,4,5};
auto even = [](int i){ return 0 == i % 2; };
auto square = [](int i) { return i * i; };
for (int i : ints | views::filter(even) | views::transform(square)) {
    std::cout << i << ' '; // prints: 0 4 16
}
assert(ranges::equal(ints | views::filter(even), views::filter(ints, even)));
```
—end example]

24.7.2 Range adaptor objects

A range adaptor closure object is a unary function object that accepts a viewable_range argument and returns a view. For a range adaptor closure object C and an expression R such that `decltype((R))` models viewable_range, the following expressions are equivalent and yield a view:

```
C(R)
R | C
```

Given an additional range adaptor closure object D, the expression C | D is well-formed and produces another range adaptor closure object such that the following two expressions are equivalent:

```
R | C | D
R | (C | D)
```

A range adaptor object is a customization point object (16.3.3.6) that accepts a viewable_range as its first argument and returns a view.

If a range adaptor object accepts only one argument, then it is a range adaptor closure object.

If a range adaptor object accepts more than one argument, then the following expressions are equivalent:

```
adaptor(range, args...)
adaptor(args...)(range)
range | adaptor(args...)
```

In this case, `adaptor(args...)` is a range adaptor closure object.

24.7.3 Semiregular wrapper

Many types in this subclause are specified in terms of an exposition-only class template `semiregular-box<T>`. `semiregular-box<T>` behaves exactly like `optional<T>` with the following differences:

1. `semiregular-box<T>` constrains its type parameter T with `copy_constructible<T> && is_object_v<T>`.
2. If T models default_initializable, the default constructor of `semiregular-box<T>` is equivalent to:

```cpp
constexpr semiregular-box() noexcept(is_nothrow_default_constructible_v<T>)
    : semiregular-box{in_place}
```

§ 24.7.3

1008
If assignable_from<T&, const T&> is not modeled, the copy assignment operator is equivalent to:

```cpp
semiregular-box& operator=(const semiregular-box& that)
    noexcept(is_nothrow_copy_constructible_v<T>)
{
    if (that) emplace(*that);
    else reset();
    return *this;
}
```

If assignable_from<T&, T> is not modeled, the move assignment operator is equivalent to:

```cpp
semiregular-box& operator=(semiregular-box&& that)
    noexcept(is_nothrow_move_constructible_v<T>)
{
    if (that) emplace(std::move(*that));
    else reset();
    return *this;
}
```

24.7.4 All view [range.all]

24.7.4.1 General [range.all.general]

1 views::all returns a view that includes all elements of its range argument.

2 The name views::all denotes a range adaptor object (24.7.2). Given a subexpression E, the expression views::all(E) is expression-equivalent to:

1 — decay-copy(E) if the decayed type of E models view.

2 — Otherwise, ref_view(E) if that expression is well-formed.

3 — Otherwise, subrange(E).

24.7.4.2 Class template ref_view [range.ref.view]

1 ref_view is a view of the elements of some other range.

```cpp
namespace std::ranges {
    template<range R>
    requires is_object_v<R>
    class ref_view : public view_interface<ref_view<R>> {
        private:
            R* r_ = nullptr; // exposition only
        public:
           constexpr ref_view() noexcept = default;
            template<not_same_as<ref_view> T>
            requires see below
            constexpr ref_view(T& t);

            constexpr R& base() const { return *r_; }

            constexpr iterator_t<R> begin() const { return ranges::begin(*r_); }
            constexpr sentinel_t<R> end() const { return ranges::end(*r_); }

            constexpr bool empty() const
                requires requires { ranges::empty(*r_); }
                { return ranges::empty(*r_); }

            constexpr auto size() const requires sized_range<R>
                requires requires { ranges::size(*r_); }
                { return ranges::size(*r_); }

            constexpr auto data() const requires contiguous_range<R>
                requires requires { ranges::data(*r_); }
                { return ranges::data(*r_); }
            );
            template<class R>
            ref_view(R&) -> ref_view<R>;
        }
    } // std::ranges

§ 24.7.4.2
template<not_same_as<ref_view> T>
   requires see below
   constexpr ref_view(T&& t);

2 Effects: Initializes r_ with addressof(static_cast<R&>(std::forward<T>(t))).

3 Remarks: Let FUN denote the exposition-only functions
   void FUN(R&);
   void FUN(R&&) = delete;

   The expression in the requires-clause is equivalent to:
   convertible_to<T, R&> && requires { FUN(declval<T>()); }

24.7.5 Filter view

24.7.5.1 Overview

1 filter_view presents a view of the elements of an underlying sequence that satisfy a predicate.

2 The name views::filter denotes a range adaptor object (24.7.2). Given subexpressions E and P, the
   expression views::filter(E, P) is expression-equivalent to filter_view(E, P).

3 [Example 1]:
   vector<int> is{ 0, 1, 2, 3, 4, 5, 6 };
   filter_view evens(is, [](int i) { return 0 == i % 2; });
   for (int i : evens)
       cout << i << ' '; // prints: 0 2 4 6
   — end example

24.7.5.2 Class template filter_view

namespace std::ranges {
   template<input_range V, indirect_unary_predicate<iterator_t<V>> Pred>
      requires view<V> && is_object_v<Pred>
   class filter_view : public view_interface<filter_view<V, Pred>> {
      private:
      V base_ = V();          // exposition only
      semiregular_box<Pred> pred_; // exposition only

      // 24.7.5.3, class filter_view::iterator
      class iterator;
      // 24.7.5.4, class filter_view::sentinel
      class sentinel;          // exposition only

      public:
      filter_view() = default;
      constexpr filter_view(V base, Pred pred);

      constexpr V base() const& requires copy_constructible<V> { return base_; }
      constexpr V base() && { return std::move(base_); }

      constexpr Pred& pred() const;

      constexpr iterator begin();
      constexpr auto end() {
         if constexpr (common_range<V>)
            return iterator{+this, ranges::end(base_.)};
         else
            return sentinel{+this};
      }
   };
   template<class R, class Pred>
   filter_view(R&&, Pred) -> filter_view<views::all_t<R>, Pred>;
}
constexpr filter_view(V base, Pred pred);

Effects: Initializes base_ with std::move(base) and initializes pred_ with std::move(pred).

constexpr const Pred& pred() const;

Effects: Equivalent to: return *pred_;

constexpr iterator begin();

Preconditions: pred_.has_value().

Returns: {*this, ranges::find_if(base_, ref(*pred_))}.

Remarks: In order to provide the amortized constant time complexity required by the range concept when filter_view models forward_range, this function caches the result within the filter_view for use on subsequent calls.

24.7.5.3 Class filter_view::iterator

namespace std::ranges {
  template<input_range V, indirect_unary_predicate<iterator_t<V>> Pred>
  requires view<V> && is_object_v<Pred>
  class filter_view<V, Pred>::iterator {
    private:
      iterator_t<V> current_ = iterator_t<V>(); // exposition only
      filter_view* parent_ = nullptr; // exposition only
    public:
      using iterator_concept = see below;
      using iterator_category = see below;
      using value_type = range_value_t<V>;
      using difference_type = range_difference_t<V>;
      iterator() = default;
      constexpr iterator(filter_view& parent, iterator_t<V> current);

      constexpr iterator_t<V> base() const &
        requires copyable<iterator_t<V>>;
      constexpr iterator_t<V> base() &&;
      constexpr range_reference_t<V> operator*() const;
      constexpr iterator_t<V> operator->() const
        requires has-arrow<iterator_t<V>> && copyable<iterator_t<V>>;
      constexpr iterator& operator++();
      constexpr void operator++(int);
      constexpr iterator operator++(int) requires forward_range<V>;
      constexpr iterator& operator--() requires bidirectional_range<V>;
      constexpr iterator operator--(int) requires bidirectional_range<V>;
      friend constexpr bool operator==(const iterator& x, const iterator& y)
        requires equality_comparable<iterator_t<V>>;
      friend constexpr range_rvalue_reference_t<V> iter_move(const iterator& i)
        noexcept(noexcept(ranges::iter_move(i.current_)));
      friend constexpr void iter_swap(const iterator& x, const iterator& y)
        noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)));
      requires indirectly_swappable<iterator_t<V>>;
    };
  }
}

1 Modification of the element a filter_view::iterator denotes is permitted, but results in undefined behavior if the resulting value does not satisfy the filter predicate.

2 iterator::iterator_concept is defined as follows:

(2.1) — If V models bidirectional_range, then iterator_concept denotes bidirectional_iterator_tag.

(2.2) — Otherwise, if V models forward_range, then iterator_concept denotes forward_iterator_tag.
iterator::iterator_category is defined as follows:

- Let \( C \) denote the type \( \text{iterator_traits<iterator_t<V>>::iterator_category} \).
- If \( C \) models \( \text{derived_from<bidirectional_iterator_tag>} \), then \( \text{iterator_category}\) denotes \( \text{bidirectional_iterator_tag} \).
- Otherwise, if \( C \) models \( \text{derived_from<forward_iterator_tag>} \), then \( \text{iterator_category}\) denotes \( \text{forward_iterator_tag} \).
- Otherwise, \( \text{iterator_category}\) denotes \( C \).

### constexpr iterator(filter_view& parent, iterator_t<V> current);

**Effects:** Initializes \( \text{current}_{\_} \) with \( \text{std::move}(current) \) and \( \text{parent}_{\_} \) with addressof(parent).

### constexpr iterator_t<V> base() const &

**Requires:** copyable<iterator_t<V>>;

**Effects:** Equivalent to: return \( \text{current}_{\_} \);

### constexpr iterator_t<V> base() &&;

**Effects:** Equivalent to: return \( \text{std::move}(current) \);

### constexpr range_reference_t<V> operator*() const;

**Effects:** Equivalent to: return \( \ast current_{\_} \);

### constexpr iterator_t<V> operator->() const

**Requires:** has-arrow<iterator_t<V>>& copyable<iterator_t<V>>;

**Effects:** Equivalent to: return \( \text{current}_{\_} \);

### constexpr iterator& operator++();

**Effects:** Equivalent to:

\[
\text{current}_{\_} = \text{ranges::find_if}(\text{std::move(++current}_{\_}), \text{ranges::end(parent}_{\_}->\text{base}_{\_}), \text{ref(*parent}_{\_}->\text{pred}_{\_}))
\]

**Effects:** Equivalent to: return *this;

### constexpr void operator++(int);

**Effects:** Equivalent to: ++*this.

### constexpr iterator operator++(int) requires forward_range<V>;

**Effects:** Equivalent to:

\[
\text{auto tmp = *this};
\]

**Effects:** Equivalent to: return tmp;

### constexpr iterator& operator--() requires bidirectional_range<V>;

**Effects:** Equivalent to:

\[
\text{do}
\begin{align*}
&\text{--current}_{\_}; \\
&\text{while (!invoke(*parent}_{\_}->\text{pred}_{\_}, *current}_{\_})
\end{align*}
\]

**Effects:** Equivalent to: return tmp;

### constexpr iterator operator--(int) requires bidirectional_range<V>;

**Effects:** Equivalent to:

\[
\text{auto tmp = *this};
\]

**Effects:** Equivalent to: ---*this;

**Effects:** Equivalent to: return tmp;

### friend constexpr bool operator==(const iterator& x, const iterator& y)

**Requires:** equality_comparable<iterator_t<V>>;

**Effects:** Equivalent to: return \( x.\text{current}_{\_} == y.\text{current}_{\_} \);
friend constexpr range_value_reference_t<V> iter_move(const iterator& i)
    noexcept(noexcept(ranges::iter_move(i.current_)));  

  Effects: Equivalent to: return ranges::iter_move(i.current_);

friend constexpr void iter_swap(const iterator& x, const iterator& y)
    noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)))
    requires indirectly_swappable<iterator_t<V>>;

  Effects: Equivalent to ranges::iter_swap(x.current_, y.current_).

24.7.5.4 Class filter_view::sentinel

namespace std::ranges {
    template<input_range V, indirect_unary_predicate<iterator_t<V>> Pred>
    requires view<V> && is_object_v<Pred>
    class filter_view<V, Pred>::sentinel {
    private:
        sentinel_t<V> end_ = sentinel_t<V>(); // exposition only
    public:
        sentinel() = default;
        constexpr explicit sentinel(filter_view& parent);
    
        sentinel_t<V> base() const;
        friend constexpr bool operator==(const iterator& x, const sentinel& y);
    };

    constexpr explicit sentinel(filter_view& parent);

      Effects: Initializes end_ with ranges::end(parent.base_).

    constexpr sentinel_t<V> base() const;

      Effects: Equivalent to: return end_; 

    friend constexpr bool operator==(const iterator& x, const sentinel& y);

      Effects: Equivalent to: return x.current_ == y.end_; 

24.7.6 Transform view

24.7.6.1 Overview

transform_view presents a view of an underlying sequence after applying a transformation function to each element.

The name views::transform denotes a range adaptor object (24.7.2). Given subexpressions E and F, the expression views::transform(E, F) is expression-equivalent to transform_view{E, F}.

[Example 1:]
    vector<int> is{ 0, 1, 2, 3, 4 }; 
    transform_view squares{is, [](int i) { return i * i; }}; 
    for (int i : squares)
        cout << i << ' '; // prints: 0 1 4 9 16 
—end example] 

24.7.6.2 Class template transform_view

namespace std::ranges {
    template<input_range V, copy_constructible F>
    requires view<V> && is_object_v<F> &&
    regular_invocable<F&[], range_reference_t<V>> &&
    can_reference<invoke_result_t<F&[], range_reference_t<V>>> 
    class transform_view : public view_interface<transform_view<V, F>> {
    private:
        // 24.7.6.3, class template transform_view::iterator 
        template<bool> struct iterator; // exposition only
        // 24.7.6.4, class template transform_view::sentinel 
        template<bool> struct sentinel; // exposition only
V base_ = V();  // exposition only
semiregular-box<F> fun_;  // exposition only

public:
transform_view() = default;
constexpr transform_view(V base, F fun);

constexpr V base() const requires copy_constructible<V> { return base_; }
constexpr V base() && { return std::move(base_); }
constexpr iterator<false> begin();
constexpr iterator<true> begin() const
    requires range<const V> &&
    regular_invocable<const F&, range_reference_t<const V>>;
constexpr sentinel<false> end();
constexpr iterator<false> end() requires common_range<V>;
constexpr sentinel<true> end() const
    requires range<const V> &&
    regular_invocable<const F&, range_reference_t<const V>>;
constexpr auto size() requires sized_range<V> { return ranges::size(base_); }
constexpr auto size() const requires sized_range<const V>
    { return ranges::size(base_); }
};

constexpr transform_view(V base, F fun);

Effects: Initializes base_ with std::move(base) and fun_ with std::move(fun).

constexpr iterator<false> begin();
Effects: Equivalent to:
    return iterator<false>{*this, ranges::begin(base_)};
constexpr iterator<true> begin() const
    requires range<const V> &&
    regular_invocable<const F&, range_reference_t<const V>>;
Effects: Equivalent to:
    return iterator<true>{*this, ranges::begin(base_)};
constexpr sentinel<false> end();
Effects: Equivalent to:
    return sentinel<false>{ranges::end(base_)};
constexpr iterator<false> end() requires common_range<V>;
Effects: Equivalent to:
    return iterator<false>{*this, ranges::end(base_)};
constexpr sentinel<true> end() const
    requires range<const V> &&
    regular_invocable<const F&, range_reference_t<const V>>;
Effects: Equivalent to:
    return sentinel<true>{ranges::end(base_)};
constexpr iterator<true> end() const
    requires common_range<const V> &&
    regular_invocable<const F&, range_reference_t<const V>>;

    Effects: Equivalent to:
    return iterator<true>{*this, ranges::end(base_)};

24.7.6.3 Class template transform_view::iterator

namespace std::ranges {
    template<input_range V, copy_constructible F>
    requires view<V> && is_object_v<F> &&
    regular_invocable<F&, range_reference_t<V>> &&
    can_reference<invoke_result_t<F&, range_reference_t<V>>>;

template<bool Const>
class transform_view<V, F>::iterator {

private:
    using Parent = // exposition only
        conditional_t<Const, const transform_view, transform_view>;
    using Base = // exposition only
        conditional_t<Const, const V, V>;
    iterator_t<Base> current_ = // exposition only
        iterator_t<Base>();
    Parent* parent_ = nullptr; // exposition only

public:
    using iterator_concept = see below;
    using iterator_category = see below;
    using value_type = remove_cvref_t<invoke_result_t<F&, range_reference_t<Base>>>;
    using difference_type = range_difference_t<Base>;

    iterator() = default;
    constexpr iterator(Parent& parent, iterator_t<Base> current);
    constexpr iterator(Parent<true> i)
        requires Const && convertible_to<iterator_t<V>, iterator_t<Base>>;

    constexpr iterator_t<Base> base() const &
        requires copyable<iterator_t<Base>>;
    constexpr iterator_t<Base> base() &&;
    constexpr decltype(auto) operator[](difference_type n) const
        requires random_access_range<Base>;

    constexpr iterator& operator++();
    constexpr iterator& operator++(int);
    constexpr iterator operator++(int) requires forward_range<Base>;

    constexpr iterator& operator--();
    constexpr iterator operator--(int) requires bidirectional_range<Base>;

    constexpr iterator& operator+=(difference_type n)
        requires random_access_range<Base>;
    constexpr iterator& operator-=(difference_type n)
        requires random_access_range<Base>;
    constexpr decltype(auto) operator[](difference_type n) const
        requires random_access_range<Base>
        { return invoke(*parent_->fun_, current_[n]); }

    friend constexpr bool operator==(const iterator& x, const iterator& y)
        requires equality_comparable<iterator_t<Base>>;

    friend constexpr bool operator<(const iterator& x, const iterator& y)
        requires random_access_range<Base>;
    friend constexpr bool operator>(const iterator& x, const iterator& y)
        requires random_access_range<Base>;
}
friend constexpr bool operator<=(const iterator& x, const iterator& y) requires random_access_range<Base>;
friend constexpr bool operator>=(const iterator& x, const iterator& y) requires random_access_range<Base>;
friend constexpr auto operator<=>(const iterator& x, const iterator& y) requires random_access_range<Base> && three_way_comparable<iterator_t<Base>>;
friend constexpr iterator operator+(iterator i, difference_type n) requires random_access_range<Base>;
friend constexpr iterator operator+(difference_type n, iterator i) requires random_access_range<Base>;
friend constexpr iterator operator-(iterator i, difference_type n) requires random_access_range<Base>;
friend constexpr difference_type operator-(const iterator& x, const iterator& y) requires random_access_range<Base>;
friend constexpr decltype(auto) iter_move(const iterator& i) noexcept(noexcept(invoke(*i.parent_->fun_, *i.current_))) {
  if constexpr (is_lvalue_reference_v<decltype(*i)>)
    return std::move(*i);
  else
    return *i;
}
friend constexpr void iter_swap(const iterator& x, const iterator& y) noexcept(noexcept(ranges::iter_swap(x.current_, y.current_))) requires indirectly_swappable<iterator_t<Base>>;

1 iterator::iterator_concept is defined as follows:
   (1.1) If V models random_access_range, then iterator_concept denotes random_access_iterator_tag.
   (1.2) Otherwise, if V models bidirectional_range, then iterator_concept denotes bidirectional_iterator_tag.
   (1.3) Otherwise, if V models forward_range, then iterator_concept denotes forward_iterator_tag.
   (1.4) Otherwise, iterator_concept denotes input_iterator_tag.

2 iterator::iterator_category is defined as follows: Let C denote the type iterator_traits<iterator_t<Base>>::iterator_category.
   (2.1) If is_lvalue_reference_v<invoke_result_t<F&, range_reference_t<Base>>> is true, then
   (2.1.1) if C models derived_from<contiguous_iterator_tag>, iterator_category denotes random_access_iterator_tag;
   (2.1.2) otherwise, iterator_category denotes C.
   (2.2) Otherwise, iterator_category denotes input_iterator_tag.

constexpr iterator(Parent& parent, iterator_t<Base> current);

   Effects: Initializes current_ with std::move(current) and parent_ with addressof(parent).

constexpr iterator(iterator<!Const> i) requires Const && convertible_to<iterator_t<Base>>;

   Effects: Initializes current_ with std::move(i.current_) and parent_ with i.parent_.

constexpr iterator_t<Base> base() const & requires copyable<iterator_t<Base>>;

   Effects: Equivalent to: return current_;

constexpr iterator_t<Base> base() &&;

   Effects: Equivalent to: return std::move(current_);
constexpr iterator operator++();

Effects: Equivalent to:

++current_;  
return *this;

constexpr void operator++(int);

Effects: Equivalent to ++current_.

constexpr iterator operator++(int) requires forward_range<Base>;

Effects: Equivalent to:

auto tmp = *this;  
++*this;  
return tmp;

constexpr iterator operator--() requires bidirectional_range<Base>;

Effects: Equivalent to:

--current_;  
return *this;

constexpr iterator operator--(int) requires bidirectional_range<Base>;

Effects: Equivalent to:

auto tmp = *this;  
--*this;  
return tmp;

constexpr iterator operator+=(difference_type n) requires random_access_range<Base>;

Effects: Equivalent to:

current_ += n;  
return *this;

constexpr iterator operator-=(difference_type n) requires random_access_range<Base>;

Effects: Equivalent to:

current_ -= n;  
return *this;

friend constexpr bool operator==(const iterator& x, const iterator& y) requires equality_comparable<iterator_t<Base>>;

Effects: Equivalent to: return x.current_ == y.current_; 

friend constexpr bool operator<(const iterator& x, const iterator& y) requires random_access_range<Base>;

Effects: Equivalent to: return x.current_ < y.current_; 

friend constexpr bool operator>(const iterator& x, const iterator& y) requires random_access_range<Base>;

Effects: Equivalent to: return y < x; 

friend constexpr bool operator<=(const iterator& x, const iterator& y) requires random_access_range<Base>;

Effects: Equivalent to: return !(y < x); 

friend constexpr bool operator>=(const iterator& x, const iterator& y) requires random_access_range<Base>;

Effects: Equivalent to: return !(x < y);
friend constexpr auto operator<=>(const iterator& x, const iterator& y) requires random_access_range<Base> & three_way_comparable<iterator_t<Base>>;

**Effects:** Equivalent to: return x.current_ <=> y.current_;

friend constexpr iterator operator+(iterator i, difference_type n) requires random_access_range<Base>;
friend constexpr iterator operator+(difference_type n, iterator i) requires random_access_range<Base>;

**Effects:** Equivalent to: return iterator{*i.parent_, i.current_ + n};

friend constexpr iterator operator-(iterator i, difference_type n) requires random_access_range<Base>;

**Effects:** Equivalent to: return iterator{*i.parent_, i.current_ - n};

friend constexpr difference_type operator-(const iterator& x, const iterator& y) requires random_access_range<Base>;

**Effects:** Equivalent to: return x.current_ - y.current_;

friend constexpr void iter_swap(const iterator& x, const iterator& y) noexcept(noexcept(ranges::iter_swap(x.current_, y.current_))) requires indirectly_swappable<iterator_t<Base>>;

**Effects:** Equivalent to ranges::iter_swap(x.current_, y.current_).

### 24.7.6.4 Class template transform_view::sentinel [range.transform.sentinel]

namespace std::ranges {
    template<input_range V, copy_constructible F> requires view<V> && is_object_v<F> &&
        regular_invocable<F&, range_reference_t<V>> &&
        can-reference<invoke_result_t<F&, range_reference_t<V>>> template<Bool Const>
    class transform_view<V, F>::sentinel {
        private:
            using Parent = // exposition only
                conditional_t<Const, const transform_view, transform_view>;
            using Base = conditional_t<Const, const V, V>; // exposition only
            sentinel_t<Base> end_ = sentinel_t<Base>(); // exposition only
        
        public:
            sentinel() = default;
            constexpr explicit sentinel(sentinel_t<Base> end);
            constexpr sentinel(sentinel_t<Const> i) requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;

            constexpr sentinel_t<Base> base() const;

            friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);

            friend constexpr range_difference_t<Base>
                operator-(const iterator<Const>& x, const sentinel& y) requires sized_sentinel_for<sentinel_t<Base>>, iterator_t<Base>>;

            friend constexpr range_difference_t<Base>
                operator-(const sentinel& y, const iterator<Const>& x) requires sized_sentinel_for<sentinel_t<Base>>, iterator_t<Base>>;
                
        }
    }
    
    constexpr explicit sentinel(sentinel_t<Base> end);
    
    **Effects:** Initializes end_ with end.

    constexpr sentinel(sentinel_t<Const> i) requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;
    
    **Effects:** Initializes end_ with std::move(i.end_).
constexpr sentinel_t<Base> base() const;

Effects: Equivalent to: return end_;
constexpr auto begin() requires (!simple-view<V>) {
  if constexpr (sized_range<V>) {
    if constexpr (random_access_range<V>)
      return ranges::begin(base_);
    else {
      auto sz = size();
      return counted_iterator{ranges::begin(base_), sz};
    }
  } else
    return counted_iterator{ranges::begin(base_), count_};
}

constexpr auto begin() const requires range<const V> {
  if constexpr (sized_range<const V>) {
    if constexpr (random_access_range<const V>)
      return ranges::begin(base_);
    else {
      auto sz = size();
      return counted_iterator{ranges::begin(base_), sz};
    }
  } else
    return counted_iterator{ranges::begin(base_), count_};
}

constexpr auto end() requires (!simple-view<V>) {
  if constexpr (sized_range<V>) {
    if constexpr (random_access_range<V>)
      return ranges::begin(base_) + size();
    else
      return default_sentinel;
  } else
    return sentinel<false>{ranges::end(base_)};
}

constexpr auto end() const requires range<const V> {
  if constexpr (sized_range<const V>) {
    if constexpr (random_access_range<const V>)
      return ranges::begin(base_) + size();
    else
      return default_sentinel;
  } else
    return sentinel<true>{ranges::end(base_)};
}

constexpr auto size() requires sized_range<V> {
  auto n = ranges::size(base_);
  return ranges::min(n, static_cast<decltype(n)>(count_));
}

constexpr auto size() const requires sized_range<const V> {
  auto n = ranges::size(base_);
  return ranges::min(n, static_cast<decltype(n)>(count_));
};

template<range R>
take_view(R&&, range_difference_t<R>)
  -> take_view<views::all_t<R>>;

constexpr take_view(V base, range_difference_t<V> count);

Effects: Initializes base_ with std::move(base) and count_ with count.
24.7.7.3 Class template take_view::sentinel

namespace std::ranges {
    template<view V>
    template<bool Const>
    class take_view<V>::sentinel {
        private:
            using Base = conditional_t<Const, const V, V>; // exposition only
            using CI = counted_iterator<iterator_t<Base>>; // exposition only
            sentinel_t<Base> end_ = sentinel_t<Base>(); // exposition only
        public:
            sentinel() = default;
            constexpr explicit sentinel(sentinel_t<Base> end);
            constexpr sentinel(sentinel<!Const> s)
                requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;
            constexpr sentinel_t<Base> base() const;
            friend constexpr bool operator==(const CI& y, const sentinel& x);
    }
}

constexpr explicit sentinel(sentinel_t<Base> end);
1 Effects: Initializes end_ with end.

constexpr sentinel(sentinel<!Const> s)
    requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;
2 Effects: Initializes end_ with std::move(s.end_).

constexpr sentinel_t<Base> base() const;
3 Effects: Equivalent to: return end_;

friend constexpr bool operator==(const CI& y, const sentinel& x);
4 Effects: Equivalent to: return y.count() == 0 || y.base() == x.end_;
V base_ = V(); // exposition only
semiregular-box<Pred> pred_; // exposition only

public:
    take_while_view() = default;
    constexpr take_while_view(V base, Pred pred);

    constexpr V base() const& requires copy_constructible<V> { return base_; }
    constexpr V base() && { return std::move(base_); }

    constexpr Pred& pred() const;
    constexpr auto begin() requires (!simple-view<V>)
    { return ranges::begin(base_); }

    constexpr auto begin() const requires range<const V>
    { return ranges::begin(base_); }

    constexpr auto end() requires (!simple-view<V>)
    { return sentinel<false>(ranges::end(base_), addressof(*pred_)); }

    constexpr auto end() const requires range<const V>
    { return sentinel<true>(ranges::end(base_), addressof(*pred_)); }

    template<class R, class Pred>
        take_while_view(R&&, Pred) -> take_while_view<views::all_t<R>, Pred>;

    template<class R, class Pred>
        take_while_view(V base, Pred pred);

Effects: Initializes base_ with std::move(base) and pred_ with std::move(pred).

constexpr Pred& pred() const;
Effects: Equivalent to:
return *pred_;

24.7.8.3 Class template take_while_view::sentinel

namespace std::ranges {
    template<view V, class Pred>
        requires input_range<V> && is_object_v<Pred> &&
        indirect_unary_predicate<const Pred, iterator_t<V>>
    template<bool Const>
        class take_while_view<V, Pred>::sentinel {
            using Base = conditional_t<Const, const V, V>;
            sentinel_t<Base> end_ = sentinel_t<Base>();
            const Pred* pred_ = nullptr;

            friend constexpr bool operator==(const iterator_t<Base>& x, const sentinel_t& y);
        };

    template<class R, class Pred>
        sentinel<sentinel_t<Base> end, const Pred* pred>;

Effects: Initializes end_ with end and pred_ with pred.

§ 24.7.8.3
constexpr sentinel(sentinel<!Const> s)
  requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;

friend constexpr bool operator==(const iterator_t<Base> & x, const sentinel & y);

Effects: Equivalent to: return y.end_ == x || !invoke(*y.pred_, *x);

24.7.9 Drop view

24.7.9.1 Overview

drop_view produces a view excluding the first \( N \) elements from another view, or an empty range if the adapted view contains fewer than \( N \) elements.

The name \texttt{views::drop} denotes a range adaptor object (24.7.2). Let \( E \) and \( F \) be expressions, let \( T \) be remove_cvref_t<decltype((E))>, and let \( D \) be range_difference_t<decltype((E))>. If \( \text{decltype}((F)) \) does not model convertible_to_t\(<D>\), \texttt{views::drop}(E, F) is ill-formed. Otherwise, the expression \texttt{views::drop}(E, F) is expression-equivalent to:

\begin{enumerate}
\item If \( T \) is a specialization of \texttt{ranges::empty_view} (24.6.2.2), then ((void) F, decay-copy(E)).
\item Otherwise, if \( T \) models \texttt{random_access_range} and \texttt{sized_range} and is
\begin{enumerate}
\item a specialization of \texttt{span} (22.7.3) where \( T::\text{extent} == \text{dynamic_extent} \),
\item a specialization of \texttt{basic_string_view} (21.4),
\item a specialization of \texttt{ranges::iota_view} (24.6.4.2), or
\item a specialization of \texttt{ranges::subrange} (24.5.4),
\end{enumerate}
then \( T\{\text{ranges::begin}(E) + \min\langle D\rangle(\text{ranges::size}(E), F), \text{ranges::end}(E)\} \), except that \( E \) is evaluated only once.
\item Otherwise, \texttt{ranges::drop_view}(E, F).
\end{enumerate}

[Example 1]:

```
auto ints = views::iota(0) | views::take(10);
auto latter_half = drop_view{ints, 5};
for (auto i : latter_half) {
  cout << i << ' ';  // prints 5 6 7 8 9
}
```

—end example—

24.7.9.2 Class template drop_view

namespace std::ranges {

  template<view V>
  class drop_view : public view_interface<drop_view<V>> {
    public:
      drop_view() = default;
      constexpr drop_view(V base, range_difference_t<V> count);
      
      constexpr V base() const \& requires copy_constructible\( V \) { return base_; }
      constexpr V base() \&\& { return std::move(base_); }
      
      constexpr auto begin()
        requires (!\texttt{simple-view}<\texttt{V} \&\& \texttt{random_access_range}<\texttt{V}>);
      constexpr auto begin() const
        requires \texttt{random_access_range}<\texttt{const V}>;
      
      constexpr auto end()
        requires (!\texttt{simple-view}<\texttt{V}>)
        { return ranges::end(base_); }
      
      constexpr auto end() const
        requires range<\texttt{const V}>
        { return ranges::end(base_); }
  }
}
constexpr auto size()
    requires sized_range<V>
{
    const auto s = ranges::size(base_);
    const auto c = static_cast<decltype(s)>(count_);
    return s < c ? 0 : s - c;
}

constexpr auto size() const
    requires sized_range<const V>
{
    const auto s = ranges::size(base_);
    const auto c = static_cast<decltype(s)>(count_);
    return s < c ? 0 : s - c;
}

private:
    V base_ = V(); // exposition only
    range_difference_t<V> count_ = 0; // exposition only
};

template<class R>
drop_view(R&&, range_difference_t<R>) -> drop_view<views::all_t<R>>;

constexpr drop_view(V base, range_difference_t<V> count);

1   Preconditions: count >= 0 is true.
2   Effects: Initializes base_ with std::move(base) and count_ with count.

constexpr auto begin()
    requires (!((simple-view<V> && random_access_range<V>)));
constexpr auto begin() const
    requires random_access_range<const V>;

3   Returns: ranges::next(ranges::begin(base_), count_, ranges::end(base_)).
4   Remarks: In order to provide the amortized constant-time complexity required by the range concept
   when drop_view models forward_range, the first overload caches the result within the drop_view for use on subsequent calls.
   [Note 1: Without this, applying a reverse_view over a drop_view would have quadratic iteration complexity.
   —end note]

24.7.10 Drop while view

24.7.10.1 Overview

1   Given a unary predicate pred and a view r, drop_while_view produces a view of the range
   [ranges::find_if_not(r, pred), ranges::end(r)].
2   The name views::drop_while denotes a range adaptor object (24.7.2). Given subexpressions E and F, the
   expression views::drop_while(E, F) is expression-equivalent to drop_while_view(E, F).
3   [Example 1:
   constexpr auto source = " \t \t \t hello there";
   auto is_invisible = []((const auto x) { return x == ' ' || x == '\t'; });
   auto skip_ws = drop_while_view(source, is_invisible);
   for (auto c : skip_ws) {
       cout << c; // prints hello there with no leading space
   }
   —end example]

24.7.10.2 Class template drop_while_view

namespace std::ranges {
    template<class V, class Pred>
    requires input_range<V> && is_object_v<Pred> &&
        indirect_unary_predicate(const Pred, iterator_t<V>)

class drop_while_view : public view_interface<drop_while_view<V, Pred>> {
  public:
  drop_while_view() = default;
  constexpr drop_while_view(V base, Pred pred);

  constexpr V base() const& requires copy_constructible<V> { return base_; }
  constexpr V base() && { return std::move(base_); }

  constexpr const Pred& pred() const;
  constexpr auto begin();
  constexpr auto end() { return ranges::end(base_); }

  private:
  V base_ = V(); // exposition only
  semiregular-box<Pred> pred_; // exposition only
};

template<class R, class Pred>
drop_while_view(R&&, Pred) -> drop_while_view<views::all_t<R>, Pred>;

constexpr V base() const& requires copy_constructible<V> { return base_; }
constexpr V base() && { return std::move(base_); }

constexpr const Pred& pred() const;
constexpr auto begin();
constexpr auto end() { return ranges::end(base_); }

// exposition only
semiregular-box<Pred> pred_; // exposition only

constexpr drop_while_view(V base, Pred pred);

Effects: Initializes base_ with std::move(base) and pred_ with std::move(pred).

constexpr const Pred& pred() const;
Effects: Equivalent to: return *pred_;

constexpr auto begin();

Returns: ranges::find_if_not(base_, cref(*pred_)).

Remarks: In order to provide the amortized constant-time complexity required by the range concept when drop_while_view models forward_range, the first call caches the result within the drop_while_view for use on subsequent calls.

[Note 1: Without this, applying a reverse_view over a drop_while_view would have quadratic iteration complexity. — end note]

24.7.11 Join view

24.7.11.1 Overview

join_view flattens a view of ranges into a view.

The name views::join denotes a range adaptor object (24.7.2). Given a subexpression E, the expression views::join(E) is expression-equivalent to join_view{E}.

[Example 1:
  vector<string> ss{"hello", " ", "world", "!"};
  join_view greeting(ss);
  for (char ch : greeting)
    cout << ch; // prints: hello world!
— end example]

24.7.11.2 Class template join_view

namespace std::ranges {
  template<input_range V>
  requires view<V> && input_range<range_reference_t<V>> &&
  (is_reference_v<range_reference_t<V>> || view<range_value_t<V>>)
  class join_view : public view_interface<join_view<V>> {
    private:
    using InnerRng = // exposition only
}
range_reference_t<V>;
// 24.7.11.3, class template join_view::iterator
template<bool Const>
struct iterator;  // exposition only
// 24.7.11.4, class template join_view::sentinel
template<bool Const>
struct sentinel;  // exposition only

V base_ = V();  // exposition only
views::all_t<InnerRng> inner_ =  // exposition only, present only when !is_reference_v<InnerRng>
views::all_t<InnerRng>();

public:
join_view() = default;
constexpr explicit join_view(V base);

constexpr V base() const& requires copy_constructible<V> { return base_; }
constexpr V base() && { return std::move(base_); }

constexpr auto begin() {
  constexpr bool use_const = simple_view<V> &&
    is_reference_v<range_reference_t<V>>;
  return iterator<use_const>{*this, ranges::begin(base_)};
}

constexpr auto begin() const
requires input_range<const V> &&
  is_reference_v<range_reference_t<const V>> {
  return iterator<true>{*this, ranges::begin(base_)};
}

constexpr auto end() {
  if constexpr (forward_range<V> &&
    is_reference_v<InnerRng> && forward_range<InnerRng> &&
    common_range<V> && common_range<InnerRng>)
    return iterator<true>{*this, ranges::end(base_)};
  else
    return sentinel<true>{*this};
}

constexpr auto end() const
requires input_range<const V> &&
  is_reference_v<range_reference_t<const V>> {
  if constexpr (forward_range<const V> &&
    is_reference_v<range_reference_t<const V>> &&
    forward_range<range_reference_t<const V>> &&
    common_range<const V> &&
    common_range<range_reference_t<const V>>) 
    return iterator<true>{*this, ranges::end(base_)};
  else
    return sentinel<true>{*this};
}

};

template<class R>
explicit join_view(R&&) -> join_view<views::all_t<R>>;

constexpr explicit join_view(V base);

1 Effects: Initializes base_ with std::move(base).

24.7.11.3 Class template join_view::iterator

namespace std::ranges {

  template<input_range V>

requires view<V> && input_range<range_reference_t<V>> &&
(is_reference_v<range_reference_t<V>> ||
 view<range_value_t<V>>)

template<bool Const>
struct join_view<V>::::iterator {
private:
    using Parent = // exposition only
conditional_t<Const, const join_view, join_view>;
    using Base = conditional_t<Const, const V, V>;
        // exposition only
static constexpr bool ref-is-glvalue =
    is_reference_v<range_reference_t<Base>>;
    iterator_t<Base> outer_ = iterator_t<Base>();
    iterator_t<range_reference_t<Base>> inner_ =
        iterator_t<range_reference_t<Base>>();
    Parent* parent_ = nullptr;
        // exposition only
    constexpr void satisfy();
        // exposition only
public:
    using iterator_concept = see below;
    using iterator_category = see below;
    using value_type = range_value_t<range_reference_t<Base>>;
    using difference_type = see below;
    iterator() = default;
    constexpr iterator(Parent& parent, iterator_t<Base> outer);
    constexpr iterator(iterator<!Const> i)
        requires Const &&
        convertible_to<iterator_t<V>, iterator_t<Base>> &&
        convertible_to<iterator_t<InnerRng>,
        iterator_t<range_reference_t<Base>>;
    constexpr decltype(auto) operator*() const { return *inner_; }
    constexpr iterator_t<Base> operator->() const
        requires has-arrow<iterator_t<Base>> && copyable<iterator_t<Base>>;
    constexpr iterator& operator++();
    constexpr void operator++(int);
    constexpr iterator operator++(int)
        requires ref-is-glvalue && forward_range<Base> &&
        forward_range<range_reference_t<Base>>;
    constexpr iterator& operator--();
    constexpr ref-is-glvalue && bidirectional_range<Base> &&
        bidirectional_range<range_reference_t<Base>> &&
        common_range<range_reference_t<Base>>;
    constexpr iterator operator--(int)
        requires ref-is-glvalue && bidirectional_range<Base> &&
        bidirectional_range<range_reference_t<Base>> &&
        common_range<range_reference_t<Base>>;
    friend constexpr bool operator==(const iterator x, const iterator y)
        requires ref-is-glvalue && equality_comparable<iterator_t<Base>> &&
        equality_comparable<iterator_t<range_reference_t<Base>>;
    friend constexpr decltype(auto) iter_move(const iterator x)
        noexcept(noexcept(ranges::iter_move(x.
        inner_))) {
        return ranges::iter_move(x.
        inner_); }
friend constexpr void iter_swap(const iterator& x, const iterator& y) noexcept(noexcept(ranges::iter_swap(x.inner_, y.inner_)));} 

1 iterator::iterator_concept is defined as follows:

(1.1) — If ref-is-glvalue is true and Base and range_reference_t<Base> each model bidirectional_range, then iterator_concept denotes bidirectional_iterator_tag.
(1.2) — Otherwise, if ref-is-glvalue is true and Base and range_reference_t<Base> each model forward_range, then iterator_concept denotes forward_iterator_tag.
(1.3) — Otherwise, iterator_concept denotes input_iterator_tag.

2 iterator::iterator_category is defined as follows:

(2.1) — Let OUTERC denote iterator_traits<iterator_t<Base>>::iterator_category, and let INNERC denote iterator_traits<iterator_t<range_reference_t<Base>>>::iterator_category.
(2.2) — If ref-is-glvalue is true and OUTERC and INNERC each model derived_from<bidirectional_iterator_tag>, iterator_category denotes bidirectional_iterator_tag.
(2.3) — Otherwise, if ref-is-glvalue is true and OUTERC and INNERC each model derived_from<forward_iterator_tag>, iterator_category denotes forward_iterator_tag.
(2.4) — Otherwise, if OUTERC and INNERC each model derived_from<input_iterator_tag>, iterator_category denotes input_iterator_tag.
(2.5) — Otherwise, iterator_category denotes output_iterator_tag.

3 iterator::difference_type denotes the type:

 ```cpp
class common_type_t<
    range_difference_t<Base>,
    range_difference_t<range_reference_t<Base>>>
```

4 join_view iterators use the satisfy function to skip over empty inner ranges.

constexpr void satisfy(); // exposition only

5 Effects: Equivalent to:

 ```cpp
auto update_inner = [this](range_reference_t<Base> x) -> auto & {
  if constexpr (ref-is-glvalue) // x is a reference
    return x;
  else
    return (parent_ -> inner_ = views::all(std::move(x)));
};

for (; outer_ != ranges::end(parent_ -> base_); ++outer_) {
  auto & inner = update_inner(*outer_);
  inner_ = ranges::begin(inner);
  if (inner_ != ranges::end(inner))
    return;
}
```

6 Effects: Initializes outer_ with std::move(outter) and parent_ with addressof(parent); then calls satisfy().

```cpp
constexpr iterator(Parent& parent, iterator_t<Base> outer);
```

7 Effects: Initializes outer_ with std::move(i.outer_), inner_ with std::move(i.inner_), and parent_ with i.parent_.
constexpr iterator_t<Base> operator->() const
requires has-arrow<iterator_t<Base>> && copyable<iterator_t<Base>>;

Effects: Equivalent to return inner_;
24.7.11.4 Class template join_view::sentinel

```cpp
namespace std::ranges {
    template<input_range V>
    requires view<V> && input_range<range_reference_t<V>> ||
    view<range_value_t<V>>
    template<bool Const>
    struct join_view<V>::sentinel {
        private:
            using Parent = conditional_t<Const, const join_view, join_view>; // exposition only
            using Base = conditional_t<Const, const V, V>; // exposition only
            sentinel_t<Base> end_ = sentinel_t<Base>(); // exposition only
        public:
            sentinel() = default;
            constexpr explicit sentinel(Parent& parent);
            constexpr sentinel(sentinel<!Const> s)
                requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;
            friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);
    };
}
```

1. **Effects:** Initializes `end_` with `ranges::end(parent.base_)`.

2. **Effects:** Initializes `end_` with `std::move(s.end_)`.

3. **Effects:** Equivalent to: return `x.outer_ == y.end_`.

24.7.12 Split view

24.7.12.1 Overview

`split_view` takes a `view` and a delimiter, and splits the `view` into subranges on the delimiter. The delimiter can be a single element or a view of elements.

2. The name `views::split` denotes a range adaptor object (24.7.2). Given subexpressions `E` and `F`, the expression `views::split(E, F)` is expression-equivalent to `split_view{E, F}`.

3. **Example 1:**

   ```cpp
   string str("the quick brown fox");
   split_view sentence(str, ' ');
   for (auto word : sentence) {
       for (char ch : word)
           cout << ch;
       cout << '*';
   }
   // The above prints: the*quick*brown*fox*
   ```

24.7.12.2 Class template split_view

```cpp
namespace std::ranges {
    template<auto> struct require-constant; // exposition only
    template<class R>
    concept tiny-range = sized_range<R> &&
        requires { typename require-constant<remove_reference_t<R>::size>(); } &&
        (remove_reference_t<R>::size() <= 1);
```
template<input_range V, forward_range Pattern>
requires view<V> && view<Pattern> &&
  indirectly_comparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to> &&
  (forward_range<V> || tiny_range<Pattern>)
class split_view : public view_interface<split_view<V, Pattern>> {
private:
  V base_ = V();                     // exposition only
  Pattern pattern_ = Pattern();     // exposition only
  iterator_t<V> current_ = iterator_t<V>(); // exposition only, present only if !forward_range<V>
  // 24.7.12.3, class template split_view::outer-iterator
  template<bool> struct outer-iterator;
  // exposition only
  // 24.7.12.5, class template split_view::inner-iterator
  template<bool> struct inner-iterator;
  // exposition only
public:
  split_view() = default;
  constexpr split_view(V base, Pattern pattern);

template<input_range R>
  requires constructible_from<V, views::all_t<R>> &&
    constructible_from<Pattern, single_view<range_value_t<R>>>
  constexpr split_view(R&& r, range_value_t<R> e);

constexpr V base() const& requires copy_constructible<V> { return base_; }
constexpr V base() && { return std::move(base_); }

constexpr auto begin() {
  if constexpr (forward_range<V>)
    return outer-iterator<simple-view<V>>{*this, ranges::begin(base_)};
  else {
    current_ = ranges::begin(base_);
    return outer-iterator<false>*{this};
  }
}

constexpr auto begin() const requires forward_range<V> && forward_range<const V> { return outer-iterator<true>*{this, ranges::begin(base_)}; }

constexpr auto end() requires forward_range<V> && common_range<V> { return outer-iterator<simple-view<V>>{*this, ranges::end(base_)}; }

constexpr auto end() const {
  if constexpr (forward_range<V> && forward_range<const V> && common_range<const V>)
    return outer-iterator<true>*{this, ranges::end(base_)};
  else
    return default_sentinel;
}

};

template<class R, class P>
  split_view(R&& r, P&&) -> split_view<views::all_t<R>, views::all_t<P>>;

template<input_range R>
  split_view(R&&, range_value_t<R>)
  -> split_view<views::all_t<R>, single_view<range_value_t<R>>>;
}

constexpr split_view(V base, Pattern pattern);

Effects: Initializes base_ with std::move(base), and pattern_ with std::move(pattern).

template<input_range R>
  requires constructible_from<V, views::all_t<R>> &&
    constructible_from<Pattern, single_view<range_value_t<R>>>
constexpr split_view(R&& r, range_value_t<R> e);

2 Effects: Initializes base_ with `views::all(std::forward<R>(r))`, and pattern_ with `single_view{std::move(e)}`.

### 24.7.12.3 Class template split_view::outer-iterator

```cpp
namespace std::ranges {
    template<input_range V, forward_range Pattern>
    requires view<V> && view<Pattern> &&
    indirectly_comparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to> &&
    (forward_range<V> || tiny_range<Pattern>)
    template<bool Const>
    struct split_view<V, Pattern>::outer-iterator {
        private:
            using Parent = // exposition only
                conditional_t<Const, const split_view, split_view>;
            using Base = // exposition only
                conditional_t<Const, const V, V>;
            Parent* parent_ = nullptr; // exposition only
            iterator_t<Base> current_;
                              // exposition only, present only if V models forward_range
            iterator_t<Base>() = private;

        public:
            using iterator_concept = conditional_t<forward_range<Base>, forward_iterator_tag, input_iterator_tag>;
            using iterator_category = input_iterator_tag;
            using value_type;
            using difference_type = range_difference_t<Base>;

            outer-iterator() = default;
            constexpr explicit outer-iterator(Parent& parent)
                requires (!forward_range<Base>);
            constexpr outer-iterator(Parent& parent, iterator_t<Base> current)
                requires forward_range<Base>;
            constexpr outer-iterator(outer-iterator<!Const> i)
                requires Const && convertible_to<iterator_t<V>, iterator_t<Base>>;

            constexpr explicit outer-iterator(const Parent& parent, range_value_t<V> e);

            operator*(const outer-iterator& x, const outer-iterator& y)
                requires forward_range<Base>;

        friend constexpr bool operator==(const outer-iterator& x, const outer-iterator& y);
        friend constexpr bool operator==(const outer-iterator& x, default_sentinel_t);
    }
}
```

1 Many of the following specifications refer to the notional member current of outer-iterator. current is equivalent to current_ if V models forward_range, and parent_->current_ otherwise.

2 Effects: Initializes parent_ with `addressof(parent)`. 

---

1 Many of the following specifications refer to the notional member current of outer-iterator. current is equivalent to current_ if V models forward_range, and parent_->current_ otherwise.

2 Effects: Initializes parent_ with `addressof(parent)`. 

---

§ 24.7.12.3 1032
```cpp
constexpr outer-iterator(Parent& parent, iterator_t<Base> current)
    requires forward_range<Base>;
3
    Effects: Initializes parent_ with addressof(parent) and current_ with std::move(current).

constexpr outer-iterator(outer-iterator<i> i)
    requires Const && convertible_to<iterator_t<T1>, iterator_t<Base>>;
4
    Effects: Initializes parent_ with i.parent_ and current_ with std::move(i.current_).

constexpr value_type operator*() const;
5
    Effects: Equivalent to: return value_type{*this};

constexpr outer-iterator& operator++();
6
    Effects: Equivalent to:
    
    const auto end = ranges::end(parent_->base_);
    if (current == end) return *this;
    const auto [pbegin, pend] = subrange{parent_->pattern_};
    if (pbegin == pend) ++current;
    else {
        do {
            auto [b, p] = ranges::mismatch(std::move(current), end, pbegin, pend);
            current = std::move(b);
            if (p == pend) { break; } // The pattern matched; skip it
        } while (++current != end);
    }
    return *this;

friend constexpr bool operator==(const outer-iterator x, const outer-iterator y)
    requires forward_range<Base>;
7
    Effects: Equivalent to: return x.current_ == y.current_;

friend constexpr bool operator==(const outer-iterator x, default_sentinel_t);
8
    Effects: Equivalent to: return x.current == ranges::end(x.parent_->base_);

24.7.12.4 Class split_view::outer-iterator::value_type

namespace std::ranges {
    template<input_range V, forward_range Pattern>
    requires view<V> && view<Pattern> &&
        indirectly_comparable<iterator_t<T1>, iterator_t<T2>, ranges::equal_to> &&
        (forward_range<V> || tiny-range<Pattern>)
    template<bool Const>
    struct split_view<V, Pattern>::outer-iterator<Const>::value_type
        : view_interface<value_type> {
        private:
            outer-iterator i_ = outer-iterator(); // exposition only
        public:
            value_type() = default;
            constexpr explicit value_type(outer-iterator i);

            constexpr inner-iterator<Const> begin() const requires copyable<outer-iterator>;
            constexpr inner-iterator<Const> begin() requires (!copyable<outer-iterator>);
            constexpr default_sentinel_t end() const;
        }
    }

constexpr explicit value_type(outer-iterator i);
1
    Effects: Initializes i_ with std::move(i).

constexpr inner-iterator<Const> begin() const requires copyable<outer-iterator>;
2
    Effects: Equivalent to: return inner-iterator<Const>{i_};
```
constexpr inner_iterator<Const> begin() requires (!copyable<outer_iterator>);

Effects: Equivalent to: return inner_iterator<Const>{std::move(i_)};

constexpr default_sentinel_t end() const;

Effects: Equivalent to: return default_sentinel;

24.7.12.5 Class template split_view::inner_iterator

namespace std::ranges {
    template<input_range V, forward_range Pattern>
    requires view<V> && view<Pattern> &&
        indirectly_comparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to> &&
        (forward_range<V> || tiny-range<Pattern>)
    struct split_view<V, Pattern>::inner_iterator {

        private:
            using Base = conditional_t<Const, const V, V>;         // exposition only
            outer_iterator<Const> i_ = outer_iterator<Const>();       // exposition only
            bool incremented_ = false;                                // exposition only

        public:
            using iterator_concept = typename outer_iterator<Const>::iterator_concept;
            using iterator_category = see below;
            using value_type = range_value_t<Base>;
            using difference_type = range_difference_t<Base>;

            inner_iterator() = default;
            constexpr explicit inner_iterator(outer_iterator<Const> i);

            constexpr decltype(auto) operator*() const { return *i_.current; }
            constexpr inner_iterator& operator++();
            constexpr decltype(auto) operator++(int) {
                if constexpr (forward_range<V>) {
                    auto tmp = *this;
                    +++this;
                    return tmp;
                } else
                    +++this;
            }

            friend constexpr bool operator==(const inner_iterator& x, const inner_iterator& y)
                requires forward_range<Base>;

            friend constexpr bool operator==(const inner_iterator& x, default_sentinel_t);

            friend constexpr decltype(auto) iter_move(const inner_iterator& i)
                noexcept(noexcept(ranges::iter_move(i_.i_.current))) {
                    return ranges::iter_move(i_.i_.current);
                }

            friend constexpr void iter_swap(const inner_iterator& x, const inner_iterator& y)
                noexcept(noexcept(ranges::iter_swap(x.i_.current, y.i_.current)))
                requires indirectly_swappable<iterator_t<Base>>;
        } // struct split_view<V, Pattern>::inner_iterator
    } // namespace std::ranges
}

1 The typedef-name iterator_category denotes:
   — forward_iterator_tag if iterator_traits<iterator_t<Base>>::iterator_category models
     derived_from<forward_iterator_tag>;
   — otherwise, iterator_traits<iterator_t<Base>>::iterator_category.

2 Effects: Initializes i_ with std::move(i).
constexpr inner-iterator & operator++();

Effects: Equivalent to:
  incremented_ = true;
  if constexpr (!forward_range<Base>) {
    if constexpr (Pattern::size() == 0) {
      return *this;
    }
  }
  ++i_.current;
  return *this;

friend constexpr bool operator==(const inner-iterator & x, const inner-iterator & y)
  requires forward_range<Base>;

Effects: Equivalent to: return x.i_.current == y.i_.current;

friend constexpr bool operator==(const inner-iterator & x, default_sentinel_t);

Effects: Equivalent to:
  auto [pcur, pend] = subrange{x.i_.parent_->pattern_};
  auto end = ranges::end(x.i_.parent_->base_);
  if constexpr (tiny-range<Pattern>) {
    const auto & cur = x.i_.current;
    if (cur == end) return true;
    if (pcur == pend) return x.incremented_;
    return *cur == *pcur;
  } else {
    auto cur = x.i_.current;
    if (cur == end) return true;
    if (pcur == pend) return x.incremented_;
    do {
      if (*cur != *pcur) return false;
      if (++pcur == pend) return true;
    } while (++cur != end);
    return false;
  }

friend constexpr void iter_swap(const inner-iterator & x, const inner-iterator & y)
  noexcept(noexcept(ranges::iter_swap(x.i_.current, y.i_.current)))
  requires indirectly_swappable<iterator_t<Base>>;

Effects: Equivalent to ranges::iter_swap(x.i_.current, y.i_.current).

24.7.13 Counted view

A counted view presents a view of the elements of the counted range (23.3.1) \(i + [0, n)\) for an iterator \(i\) and non-negative integer \(n\).

The name views::counted denotes a customization point object (16.3.3.3.6). Let \(E\) and \(F\) be expressions, let \(T\) be decay_t<decltype((E))>, and let \(D\) be iter_difference_t<T>. If decltype((F)) does not model convertible_to<\(D\), views::counted(E, F) is ill-formed.

[Note 1: This case can result in substitution failure when views::counted(E, F) appears in the immediate context of a template instantiation. — end note]

Otherwise, views::counted(E, F) is expression-equivalent to:

(2.1) If \(T\) models contiguous_iterator, then span(to_address(E), static_cast<\(D\)>(F)).

(2.2) Otherwise, if \(T\) models random_access_iterator, then subrange(E, E + static_cast<\(D\)>(F)), except that \(E\) is evaluated only once.

(2.3) Otherwise, subrange{counted_iterator(E, F), default_sentinel}.
24.7.14 Common view

24.7.14.1 Overview

`common_view` takes a view which has different types for its iterator and sentinel and turns it into a view of the same elements with an iterator and sentinel of the same type.

[Note 1: `common_view` is useful for calling legacy algorithms that expect a range's iterator and sentinel types to be the same. — end note]

The name `views::common` denotes a range adaptor object (24.7.2). Given a subexpression `E`, the expression `views::common(E)` is expression-equivalent to:

1. `views::all(E), if decltype((E)) models common_range and views::all(E) is a well-formed expression.
2. Otherwise, `common_view{E}`.

[Example 1:
```
// Legacy algorithm:
template<class ForwardIterator>
size_t count(ForwardIterator first, ForwardIterator last);

template<forward_range R>
void my_algo(R&& r) {
  auto&& common = common_view{r};
  auto cnt = count(common.begin(), common.end());
  // ...
}
```
—end example]

24.7.14.2 Class template `common_view`

```
namespace std::ranges {

  template<view V>
  requires (!common_range<V> && copyable<iterator_t<V>>)
  class common_view : public view_interface<common_view<V>> {
    private:
      V base_ = V(); // exposition only
    public:
      constexpr explicit common_view(V r);
    
    template<viewable_range R>
    requires (!common_range<R> && constructible_from<V, views::all_t<R>>)
    constexpr explicit common_view(R&& r);

    constexpr V base() const& requires copy_constructible<V> { return base_; }
    constexpr V base() && { return std::move(base_); }

    constexpr auto begin() {
      if constexpr (random_access_range<V> && sized_range<V>)
        return ranges::begin(base_);
      else
        return common_iterator<iterator_t<V>, sentinel_t<V>>(ranges::begin(base_));
    }

    constexpr auto begin() const requires range<const V> {
      if constexpr (random_access_range<const V> && sized_range<const V>)
        return ranges::begin(base_);
      else
        return common_iterator<iterator_t<const V>, sentinel_t<const V>>(ranges::begin(base_));
    }

    constexpr auto end() {
      if constexpr (random_access_range<V> && sized_range<V>)
        return ranges::begin(base_) + ranges::size(base_);
    }

```

§ 24.7.14.2 1036
```cpp
else
    return common_iterator<iterator_t<V>, sentinel_t<V>>(ranges::end(base_));
}

constexpr auto end() const requires range<const V> {
    if constexpr (random_access_range<const V> && sized_range<const V>)
        return ranges::begin(base_) + ranges::size(base_);
    else
        return common_iterator<iterator_t<const V>, sentinel_t<const V>>(ranges::end(base_));
}

constexpr auto size() requires sized_range<V> {
    return ranges::size(base_);
}

constexpr auto size() const requires sized_range<const V> {
    return ranges::size(base_);
};

template<class R>
common_view(R&&) -> common_view<views::all_t<R>>;

constexpr explicit common_view(V base);

Effects: Initializes base_ with std::move(base).

template<viewable_range R>
requires (!common_range<R> && constructible_from<V, views::all_t<R>>)
constexpr explicit common_view(R&& r);

Effects: Initializes base_ with views::all(std::forward<R>(r)).

24.7.15 Reverse view

24.7.15.1 Overview

reverse_view takes a bidirectional view and produces another view that iterates the same elements in reverse order.

The name views::reverse denotes a range adaptor object (24.7.2). Given a subexpression E, the expression views::reverse(E) is expression-equivalent to:

(2.1) — If the type of E is a (possibly cv-qualified) specialization of reverse_view, equivalent to E.base().
(2.2) — Otherwise, if the type of E is cv subrange<reverse_iterator<I>, reverse_iterator<I>, K> for some iterator type I and value K of type subrange_kind,

(2.2.1) — if K is subrange_kind::sized, equivalent to:
    subrange<I, I, K>(E.end().base(), E.begin().base(), E.size())
(2.2.2) — otherwise, equivalent to:
    subrange<I, I, K>(E.end().base(), E.begin().base())

However, in either case E is evaluated only once.

(2.3) — Otherwise, equivalent to reverse_view(E).

Example 1:

vector<int> is {0,1,2,3,4};
reverse_view rv {is};
for (int i : rv)
    cout << i << ' '; // prints: 4 3 2 1 0
end example]

24.7.15.2 Class template reverse_view

namespace std::ranges {
    template<view V>
    requires bidirectional_range<V>
```
class reverse_view : public view_interface<reverse_view<V>> {
private:
    V base_ = V(); // exposition only
public:
    reverse_view() = default;

    constexpr explicit reverse_view(V r);
    constexpr V base() const requires copy_constructible<V> { return base_; }
    constexpr V base() && { return std::move(base_); }
    constexpr reverse_iterator<iterator_t<V>> begin();
    constexpr reverse_iterator<iterator_t<V>> begin() requires common_range<V>;
    constexpr auto begin() const requires common_range<const V>;
    constexpr reverse_iterator<iterator_t<V>> end();
    constexpr auto end() const requires common_range<const V>;
    constexpr auto size() requires sized_range<V> {
        return ranges::size(base_);
    }
    constexpr auto size() const requires sized_range<const V> {
        return ranges::size(base_);
    }
};

template<class R>
reverse_view(R&&) -> reverse_view<views::all_t<R>>;

constexpr explicit reverse_view(V base);
1 Effects: Initializes base_ with std::move(base).

constexpr reverse_iterator<iterator_t<V>> begin();
2 Returns:

    make_reverse_iterator(ranges::next(ranges::begin(base_), ranges::end(base_)))

3 Remarks: In order to provide the amortized constant time complexity required by the range concept, this function caches the result within the reverse_view for use on subsequent calls.

constexpr reverse_iterator<iterator_t<V>> begin() requires common_range<V>;
constexpr auto begin() const requires common_range<const V>;
4 Effects: Equivalent to: return make_reverse_iterator(ranges::end(base_));

constexpr reverse_iterator<iterator_t<V>> end();
constexpr auto end() const requires common_range<const V>;
5 Effects: Equivalent to: return make_reverse_iterator(ranges::begin(base_));

24.7.16 Elements view

24.7.16.1 Overview

1 elements_view takes a view of tuple-like values and a size_t, and produces a view with a value-type of the Nth element of the adapted view’s value-type.

2 The name views::elements<N> denotes a range adaptor object (24.7.2). Given a subexpression E and constant expression N, the expression views::elements<N>(E) is expression-equivalent to elements_view<views::all_t<decltype((E))>, N>(E).

[Example 1:]

    auto historical_figures = map{
        ("Lovelace"sv, 1815),
        ("Turing"sv, 1912),
        ("Babbage"sv, 1791),
    }
auto names = historical_figures | views::elements<0>;
for (auto&& name : names) {
    cout << name << ' '; // prints Babbage Hamilton Lovelace Turing
}

auto birth_years = historical_figures | views::elements<1>;
for (auto&& born : birth_years) {
    cout << born << ' '; // prints 1791 1936 1815 1912
}

3 keys_view is an alias for elements_view<views::all_t<R>, 0>, and is useful for extracting keys from associative containers.

[Example 2:
    auto names = keys_view<historical_figures>;
    for (auto&& name : names) {
        cout << name << ' '; // prints Babbage Hamilton Lovelace Turing
    }
  — end example]

4 values_view is an alias for elements_view<views::all_t<R>, 1>, and is useful for extracting values from associative containers.

[Example 3:
    auto is_even = [](const auto x) { return x % 2 == 0; }; 
    cout << ranges::count_if(values_view<historical_figures>, is_even); // prints 2
  — end example]

24.7.16.2 Class template elements_view

namespace std::ranges {
    template<class T, size_t N>
    concept has-tuple-element =
        // exposition only
        requires(T t) {
            typename tuple_size<T>::type;
            requires N < tuple_size_v<T>;
            typename tuple_element_t<N, T>;
            { get<N>(t) } -> convertible_to<const tuple_element_t<N, T>&>;
        }

    template<input_range V, size_t N>
    requires view<V> && has-tuple-element<range_value_t<V>, N> &&
    has-tuple-element<remove_reference_t<range_reference_t<V>>, N>
    class elements_view : public view_interface<elements_view<V, N>> {
    public:
        elements_view() = default;
        constexpr explicit elements_view(V base);
        constexpr V base() const requires copy_constructible<V> { return base_; }
        constexpr V base() && { return std::move(base_); }
        constexpr auto begin() requires (!simple_view<V>)
        { return iterator<false>(ranges::begin(base_)); }
        constexpr auto begin() const requires simple_view<V>
        { return iterator<true>(ranges::begin(base_)); }
        constexpr auto end()
        { return sentinel<false>(ranges::end(base_)); }

§ 24.7.16.2 1039
constexpr auto end() requires common_range<V>  
{ return iterator<false>{ranges::end(base_)}; }  

constexpr auto end() const requires range<const V>  
{ return sentinel<true>{ranges::end(base_)}; }  

constexpr auto end() const requires common_range<const V>  
{ return iterator<true>{ranges::end(base_)}; }  

constexpr auto size() requires sized_range<V>  
{ return ranges::size(base_); }  

constexpr auto size() const requires sized_range<const V>  
{ return ranges::size(base_); }  

private:  
  // 24.7.16.3, class template elements_view::iterator  
  template<bool> struct iterator;  // exposition only
  // 24.7.16.4, class template elements_view::sentinel  
  template<bool> struct sentinel;  // exposition only  
  V base_ = V();  // exposition only
};

constexpr explicit elements_view(V base);  

Effects: Initializes base_ with std::move(base).

24.7.16.3 Class template elements_view::iterator
[range.elements.iterator]
namespace std::ranges {
template<input_range V, size_t N>  
requires view<V> &&  
has-tuple-element<range_value_t<V>, N> &&  
has-tuple-element<remove_reference_t<range_reference_t<V>>, N>  
template<bool Const>  
class elements_view<V, N>::iterator {  // exposition only
  using Base = conditional_t<Const, const V, V>;  // exposition only
  iterator_t<Base> current_ = iterator_t<Base>();  // exposition only

  using iterator_category = typename iterator_traits<iterator_t<Base>>::iterator_category;
  using value_type = remove_cvref_t<tuple_element_t<N, range_value_t<Base>>;
  using difference_type = range_difference_t<Base>;

  iterator() = default;
  constexpr explicit iterator(iterator_t<Base> current);
  constexpr iterator(iterator<!Const> i)  
    requires Const && convertible_to<iterator_t<V>, iterator_t<Base>>;

  constexpr iterator_t<Base> base() const&  
    requires copyable<iterator_t<Base>>;
  constexpr iterator_t<Base> base() &&;

  constexpr decltype(auto) operator*() const  
    { return get<N>(*current_); }  

  constexpr iterator& operator++();
  constexpr void operator++(int) requires (!forward_range<Base>);
  constexpr iterator operator++(int) requires forward_range<Base>;

  constexpr iterator& operator--() requires bidirectional_range<Base>;
  constexpr iterator operator--(int) requires bidirectional_range<Base>;

  constexpr iterator& operator+=(difference_type x)  
    requires random_access_range<Base>;
};

§ 24.7.16.3
constexpr iterator& operator-=(difference_type x)
  requires random_access_range<Base>;

define(operator[](difference_type n) const
  requires random_access_range<Base>
  { return get<N>(*(_current_ + n)); })

friend constexpr bool operator==(const iterator& x, const iterator& y)
  requires equality_comparable<iterator_t<Base>>;

friend constexpr bool operator<(const iterator& x, const iterator& y)
  requires random_access_range<Base>;

friend constexpr bool operator>(const iterator& x, const iterator& y)
  requires random_access_range<Base>;

friend constexpr bool operator<=(const iterator& x, const iterator& y)
  requires random_access_range<Base>;

friend constexpr bool operator>=(const iterator& x, const iterator& y)
  requires random_access_range<Base>;

friend constexpr auto operator<=>(const iterator& x, const iterator& y)
  requires random_access_range<Base> && three_way_comparable<iterator_t<Base>>;

friend constexpr iterator operator+(const iterator& x, difference_type y)
  requires random_access_range<Base>;

friend constexpr iterator operator+(difference_type x, const iterator& y)
  requires random_access_range<Base>;

friend constexpr iterator operator-(const iterator& x, difference_type y)
  requires random_access_range<Base>;

friend constexpr difference_type operator-(const iterator& x, const iterator& y)
  requires random_access_range<Base>;

constexpr explicit iterator(iterator_t<Base> current);
  
  **Effects:** Initializes _current_ with std::move(current).

constexpr iterator(iterator<Const> i)
  requires Const && convertible_to<iterator_t<Base>>;

  **Effects:** Initializes _current_ with std::move(i._current_).

constexpr iterator_t<Base> base() const
  requires copyable<iterator_t<Base>>;

  **Effects:** Equivalent to: return _current_;

constexpr iterator_t<Base> base() &&;

  **Effects:** Equivalent to: return std::move(_current_);

constexpr iterator& operator++();

  **Effects:** Equivalent to:
  
  ++_current_;
  return *this;

constexpr void operator++(int) requires (!forward_range<Base>);

  **Effects:** Equivalent to: ++_current_.

constexpr iterator operator++(int) requires forward_range<Base>;

  **Effects:** Equivalent to:
  
  auto temp = *this;
  ++_current_;
  return temp;
constexpr iterator operator--() requires bidirectional_range<Base>;

**Effects**: Equivalent to:

```cpp
--current_;  
return *this;
```

constexpr iterator operator--(int) requires bidirectional_range<Base>;

**Effects**: Equivalent to:

```cpp
auto temp = *this;  
--current_;  
return temp;
```

constexpr iterator operator+=(difference_type n);

**Requires**: random_access_range<Base>;

**Effects**: Equivalent to:

```cpp
current_ += n;  
return *this;
```

constexpr iterator operator-=(difference_type n)

**Requires**: random_access_range<Base>;

**Effects**: Equivalent to:

```cpp
current_ -= n;  
return *this;
```

friend constexpr bool operator==(const iterator& x, const iterator& y)

**Requires**: equality_comparable<Base>;

**Effects**: Equivalent to:

```cpp
return x.current_ == y.current_;```

friend constexpr bool operator<(const iterator& x, const iterator& y)

**Requires**: random_access_range<Base>;

**Effects**: Equivalent to:

```cpp
return x.current_ < y.current_;```

friend constexpr bool operator>(const iterator& x, const iterator& y)

**Requires**: random_access_range<Base>;

**Effects**: Equivalent to:

```cpp
return y < x;
```

friend constexpr bool operator<=(const iterator& x, const iterator& y)

**Requires**: random_access_range<Base>;

**Effects**: Equivalent to:

```cpp
return !(y < x);
```

friend constexpr bool operator>=(const iterator& x, const iterator& y)

**Requires**: random_access_range<Base>;

**Effects**: Equivalent to:

```cpp
return !(x < y);
```

friend constexpr auto operator<=>(const iterator& x, const iterator& y)

**Requires**: random_access_range<Base> & three_way_comparable<iterator_t<Base>>;

**Effects**: Equivalent to:

```cpp
return x.current_ <=> y.current_;```

friend constexpr iterator operator+(const iterator& x, difference_type y)

**Requires**: random_access_range<Base>;

**Effects**: Equivalent to:

```cpp
return iterator{x} += y;
```

friend constexpr iterator operator+(difference_type x, const iterator& y)

**Requires**: random_access_range<Base>;

**Effects**: Equivalent to:

```cpp
return y + x;
```

friend constexpr iterator operator-(const iterator& x, difference_type y)

**Requires**: random_access_range<Base>;

**Effects**: Equivalent to:

```cpp
return iterator{x} -= y;
```
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
requires random_access_range<Base>;

   Effects: Equivalent to: return x.current_ - y.current_;

24.7.16.4 Class template elements_view::sentinel

namespace std::ranges {

  template<input_range V, size_t N>
  requires view<V> && has-tuple-element<range_value_t<V>, N> &&
  has-tuple-element<remove_reference_t<range_reference_t<V>>, N>
  template<bool Const>
  class elements_view<V, N>::sentinel {
    private:
      using Base = conditional_t<Const, const V, V>;
      sentinel_t<Base> end_ = sentinel_t<Base>();
    public:
      sentinel() = default;
      constexpr explicit sentinel(sentinel_t<Base> end);
      constexpr sentinel(sentinel<!Const> other) requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;
      constexpr sentinel_t<Base> base() const;
      friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);
      friend constexpr range_difference_t<Base>
      operator-(const iterator<Const>& x, const sentinel& y) requires sized_sentinel_for<sentinel_t<Base>, iterator_t<Base>>;
    }
  }

  constexpr explicit sentinel(sentinel_t<Base> end);

  friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);
  friend constexpr range_difference_t<Base>
  operator-(const iterator<Const>& x, const sentinel& y) requires sized_sentinel_for<sentinel_t<Base>, iterator_t<Base>>;

  friend constexpr range_difference_t<Base>
  operator-(const sentinel& x, const iterator<Const>& y) requires sized_sentinel_for<sentinel_t<Base>, iterator_t<Base>>;

  friend constexpr range_difference_t<Base>
  operator-(const iterator<Const>& x, const sentinel& y) requires sized_sentinel_for<sentinel_t<Base>, iterator_t<Base>>;

  Effects: Equivalent to: return x.current_ - y.current_;
25 Algorithms library

25.1 General

1 This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 22) and other sequences.

2 The following subclauses describe components for non-modifying sequence operations, mutating sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 91.

<table>
<thead>
<tr>
<th>Subclause Header</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.2 Algorithms requirements</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.3 Parallel algorithms</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.5 Algorithm result types</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.6 Non-modifying sequence operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.7 Mutating sequence operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.8 Sorting and related operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.10 Generalized numeric operations</td>
<td>&lt;numeric&gt;</td>
</tr>
<tr>
<td>25.11 Specialized algorithms</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>25.12 C library algorithms</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

25.2 Algorithms requirements

1 All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

2 The entities defined in the std::ranges namespace in this Clause are not found by argument-dependent name lookup (6.5.3). When found by unqualified (6.5.2) name lookup for the postfix-expression in a function call (7.6.1.3), they inhibit argument-dependent name lookup.

[Example 1:]

```cpp
void foo() {
    using namespace std::ranges;
    std::vector<int> vec{1, 2, 3};
    find(begin(vec), end(vec), 2);  // #1
}
```

The function call expression at #1 invokes std::ranges::find, not std::find, despite that (a) the iterator type returned from begin(vec) and end(vec) may be associated with namespace std and (b) std::find is more specialized (13.7.7.3) than std::ranges::find since the former requires its first two parameters to have the same type. — end example]

3 For purposes of determining the existence of data races, algorithms shall not modify objects referenced through an iterator argument unless the specification requires such modification.

4 Throughout this Clause, where the template parameters are not constrained, the names of template parameters are used to express type requirements.

(4.1) — If an algorithm’s template parameter is named InputIterator, InputIterator1, or InputIterator2, the template argument shall meet the Cpp17InputIterator requirements (23.3.5.3).

(4.2) — If an algorithm’s template parameter is named OutputIterator, OutputIterator1, or OutputIterator2, the template argument shall meet the Cpp17OutputIterator requirements (23.3.5.4).

(4.3) — If an algorithm’s template parameter is named ForwardIterator, ForwardIterator1, or ForwardIterator2, the template argument shall meet the Cpp17ForwardIterator requirements (23.3.5.5).

(4.4) — If an algorithm’s template parameter is named NoThrowForwardIterator, the template argument shall meet the Cpp17ForwardIterator requirements (23.3.5.5), and is required to have the property that no
If an algorithm’s template parameter is named `BidirectionalIterator`, `BidirectionalIterator1`, or `BidirectionalIterator2`, the template argument shall meet the `Cpp17BidirectionalIterator` requirements (23.3.5.6).

If an algorithm’s template parameter is named `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the template argument shall meet the `Cpp17RandomAccessIterator` requirements (23.3.5.7).

5 If an algorithm’s `Effects` element specifies that a value pointed to by any iterator passed as an argument is modified, then that algorithm has an additional type requirement: The type of that argument shall meet the requirements of a mutable iterator (23.3).

[Note 1: This requirement does not affect arguments that are named `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, because output iterators must always be mutable, nor does it affect arguments that are constrained, for which mutability requirements are expressed explicitly. — end note]

6 Both in-place and copying versions are provided for certain algorithms. When such a version is provided for `algorithm` it is called `algorithm_copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

7 When not otherwise constrained, the `Predicate` parameter is used whenever an algorithm expects a function object (20.14) that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument with value type `T`, it should work correctly in the construct `pred(*first)` contextually converted to `bool` (7.3). The function object `pred` shall not apply any non-constant function through the dereferenced iterator. Given a glvalue `u` of type (possibly `const`) `T` that designates the same object as `*first`, `pred(u)` shall be a valid expression that is equal to `pred(*first)`.

When not otherwise constrained, the `BinaryPredicate` parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value testable as `true`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments with respective value types `T1` and `T2`, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (7.3). Unless otherwise specified, `BinaryPredicate` always takes the first iterator’s `value_type` as its first argument, that is, in those cases when `T` value is part of the signature, it should work correctly in the construct `binary_pred(*first1, value)` contextually converted to `bool` (7.3). `binary_pred` shall not apply any non-constant function through the dereferenced iterators. Given a glvalue `u` of type (possibly `const`) `T1` that designates the same object as `*first1`, and a glvalue `v` of type (possibly `const`) `T2` that designates the same object as `*first2`, `binary_pred(u, *first2)`, `binary_pred(*first1, v)`, and `binary_pred(u, v)` shall each be a valid expression that is equal to `binary_pred(*first1, *first2)`, and `binary_pred(u, value)` shall be a valid expression that is equal to `binary_pred(*first1, value)`.

9 The parameters `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, and `BinaryOperation2` are used whenever an algorithm expects a function object (20.14).

10 [Note 2: Unless otherwise specified, algorithms that take function objects as arguments can copy those function objects freely. If object identity is important, a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>` (20.14.6), or some equivalent solution, can be used. — end note]

11 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either `true` or `false` in boolean contexts.

12 In the description of the algorithms, operator `+` is used for some of the iterator categories for which it does not have to be defined. In these cases the semantics of `a + n` are the same as those of

```c++
auto tmp = a;
for (; n < 0; ++n) --tmp;
for (; n > 0; --n) ++tmp;
return tmp;
```
Similarly, operator - is used for some combinations of iterators and sentinel types for which it does not have to be defined. If \([a, b)\) denotes a range, the semantics of \(b - a\) in these cases are the same as those of

\[
\text{iter_difference_t<typename(a)> n = 0;}
\]

\[
\text{for (auto tmp = a; tmp != b; ++tmp) ++n;}
\]

\[
\text{return n;}
\]

and if \([b, a)\) denotes a range, the same as those of

\[
\text{iter_difference_t<typename(b)> n = 0;}
\]

\[
\text{for (auto tmp = b; tmp != a; ++tmp) --n;}
\]

\[
\text{return n;}
\]

In the description of algorithm return values, a sentinel value \(s\) denoting the end of a range \([i, s)\) is sometimes returned where an iterator is expected. In these cases, the semantics are as if the sentinel is converted into an iterator using \(\text{ranges::next}(i, s)\).

Overloads of algorithms that take range arguments (24.4.2) behave as if they are implemented by calling \(\text{ranges::begin}\) and \(\text{ranges::end}\) on the range(s) and dispatching to the overload in namespace \(\text{ranges}\) that takes separate iterator and sentinel arguments.

The number and order of deducible template parameters for algorithm declarations are unspecified, except where explicitly stated otherwise.

\[\text{Note 3: Consequently, an implementation can reject calls that specify an explicit template argument list. — end note}\]

25.3 Parallel algorithms

25.3.1 Preamble

Subclause 25.3 describes components that C++ programs may use to perform operations on containers and other sequences in parallel.

A parallel algorithm is a function template listed in this document with a template parameter named ExecutionPolicy.

Parallel algorithms access objects indirectly accessible via their arguments by invoking the following functions:

- All operations of the categories of the iterators that the algorithm is instantiated with.
- Operations on those sequence elements that are required by its specification.
- User-provided function objects to be applied during the execution of the algorithm, if required by the specification.
- Operations on those function objects required by the specification.

\[\text{Note 1: See 25.2. — end note}\]

These functions are herein called element access functions.

\[\text{Example 1: The sort function may invoke the following element access functions:}\]

- Operations of the random-access iterator of the actual template argument (as per 23.3.5.7), as implied by the name of the template parameter \(\text{RandomAccessIterator}\).
- The swap function on the elements of the sequence (as per the preconditions specified in 25.8.2.1).
- The user-provided \(\text{Compare}\) function object.

\[\text{— end example}\]

A standard library function is vectorization-unsafe if it is specified to synchronize with another function invocation, or another function invocation is specified to synchronize with it, and if it is not a memory allocation or deallocation function.

\[\text{Note 2: Implementations must ensure that internal synchronization inside standard library functions does not prevent forward progress when those functions are executed by threads of execution with weakly parallel forward progress guarantees. — end note}\]

\[\text{Example 2:}\]

\[
\text{int x = 0;}
\]\n
\[
\text{std::mutex m;}
\]\n
\[
\text{void f() \{}\]

\[
\text{int a[] = \{1,2\};}
\]\n
\[\text{— end example}\]

§ 25.3.1
The above program may result in two consecutive calls to `m.lock()` on the same thread of execution (which may deadlock), because the applications of the function object are not guaranteed to run on different threads of execution.

### 25.3.2 Requirements on user-provided function objects

Unless otherwise specified, function objects passed into parallel algorithms as objects of type `Predicate`, `BinaryPredicate`, `Compare`, `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, `BinaryOperation2`, and the operators used by the analogous overloads to these parallel algorithms that are formed by an invocation with the specified default predicate or operation (where applicable) shall not directly or indirectly modify objects via their arguments, nor shall they rely on the identity of the provided objects.

### 25.3.3 Effect of execution policies on algorithm execution

Parallel algorithms have template parameters named `ExecutionPolicy` (20.18) which describe the manner in which the execution of these algorithms may be parallelized and the manner in which they apply the element access functions.

1. If an object is modified by an element access function, the algorithm will perform no other unsynchronized accesses to that object. The modifying element access functions are those which are specified as modifying the object.

   [Note 1: For example, `swap`, `++`, `--`, `@=`, and assignments modify the object. For the assignment and `@=` operators, only the left argument is modified. — end note]

2. Unless otherwise stated, implementations may make arbitrary copies of elements (with type `T`) from sequences where `is_trivially_copy_constructible_v<T>` and `is_trivially_destructible_v<T>` are `true`.

   [Note 2: This implies that user-supplied function objects cannot rely on object identity of arguments for such input sequences. If object identity of the arguments to these function objects is important, a wrapping iterator that returns a non-copied implementation object such as `reference_wrapper<T>` (20.14.6), or some equivalent solution, can be used. — end note]

3. The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `std::execution::sequenced_policy` all occur in the calling thread of execution.

   [Note 3: The invocations are not interleaved; see 6.9.1. — end note]

4. The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `std::execution::unsequenced_policy` are permitted to execute in an unordered fashion in the calling thread of execution, unsequenced with respect to one another in the calling thread of execution.

   [Note 4: This means that multiple function object invocations can be interleaved on a single thread of execution, which overrides the usual guarantee from 6.9.1 that function executions do not overlap with one another. — end note]

5. The behavior of a program is undefined if it invokes a vectorization-unsafe standard library function from user code called from a `std::execution::unsequenced_policy` algorithm.

   [Note 5: Because `std::execution::unsequenced_policy` allows the execution of element access functions to be interleaved on a single thread of execution, blocking synchronization, including the use of mutexes, risks deadlock. — end note]

6. The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `std::execution::parallel_policy` are permitted to execute either in the invoking thread of execution or in a thread of execution implicitly created by the library to support parallel algorithm execution. If the threads of execution created by `thread` (32.4.3) or `jthread` (32.4.4) provide concurrent forward progress guarantees (6.9.2.3), then a thread of execution implicitly created by the library will provide parallel forward progress guarantees; otherwise, the provided forward progress guarantee is implementation-defined. Any such invocations executing in the same thread of execution are indeterminately sequenced with respect to each other.

   [Note 6: It is the caller’s responsibility to ensure that the invocation does not introduce data races or deadlocks. — end note]
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [&] (int i) {
    v.push_back(i*2+1); // incorrect: data race
});

The program above has a data race because of the unsynchronized access to the container v. — end example]

[Example 2:

std::atomic<int> x{0};
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&] (int) {
    x.fetch_add(1, std::memory_order::relaxed);
    // spin wait for another iteration to change the value of x
    while (x.load(std::memory_order::relaxed) == 1) {} // incorrect: assumes execution order
});

The above example depends on the order of execution of the iterations, and will not terminate if both iterations are executed sequentially on the same thread of execution. — end example]

[Example 3:

int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&] (int) {
    std::lock_guard<mutex> guard(m);
    ++x;
});

The above example synchronizes access to object x ensuring that it is incremented correctly. — end example]

7 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type execution::parallel_unsequenced_policy are permitted to execute in an unordered fashion in unspecified threads of execution, and unsequenced with respect to one another within each thread of execution. These threads of execution are either the invoking thread of execution or threads of execution implicitly created by the library; the latter will provide weakly parallel forward progress guarantees.

[Note 7: This means that multiple function object invocations can be interleaved on a single thread of execution, which overrides the usual guarantee from 6.9.1 that function executions do not overlap with one another. — end note]

The behavior of a program is undefined if it invokes a vectorization-unsafe standard library function from user code called from a execution::parallel_unsequenced_policy algorithm.

[Note 8: Because execution::parallel_unsequenced_policy allows the execution of element access functions to be interleaved on a single thread of execution, blocking synchronization, including the use of mutexes, risks deadlock. — end note]

8 [Note 9: The semantics of invocation with execution::unsequenced_policy, execution::parallel_policy, or execution::parallel_unsequenced_policy allow the implementation to fall back to sequential execution if the system cannot parallelize an algorithm invocation, e.g., due to lack of resources. — end note]

9 If an invocation of a parallel algorithm uses threads of execution implicitly created by the library, then the invoking thread of execution will either

(9.1) temporarily block with forward progress guarantee delegation (6.9.2.3) on the completion of these library-managed threads of execution, or

(9.2) eventually execute an element access function;

the thread of execution will continue to do so until the algorithm is finished.

[Note 10: In blocking with forward progress guarantee delegation in this context, a thread of execution created by the library is considered to have finished execution as soon as it has finished the execution of the particular element access function that the invoking thread of execution logically depends on. — end note]

10 The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type are implementation-defined.

25.3.4 Parallel algorithm exceptions [algorithms.parallel.exceptions]

1 During the execution of a parallel algorithm, if temporary memory resources are required for parallelization and none are available, the algorithm throws a bad_alloc exception.
During the execution of a parallel algorithm, if the invocation of an element access function exits via an uncaught exception, the behavior is determined by the **ExecutionPolicy**.

25.3.5 **ExecutionPolicy algorithm overloads**

Parallel algorithms are algorithm overloads. Each parallel algorithm overload has an additional template type parameter named `ExecutionPolicy`, which is the first template parameter. Additionally, each parallel algorithm overload has an additional function parameter of type `ExecutionPolicy&&`, which is the first function parameter.

[Note 1: Not all algorithms have parallel algorithm overloads. — end note]

2 Unless otherwise specified, the semantics of `ExecutionPolicy` algorithm overloads are identical to their overloads without.

3 Unless otherwise specified, the complexity requirements of `ExecutionPolicy` algorithm overloads are relaxed from the complexity requirements of the overloads without as follows: when the guarantee says “at most `expr`” or “exactly `expr`” and does not specify the number of assignments or swaps, and `expr` is not already expressed with $O()$ notation, the complexity of the algorithm shall be $O(expr)$.

4 Parallel algorithms shall not participate in overload resolution unless `is_execution_policy_v<remove_cvref_t<ExecutionPolicy>>` is true.

25.4 **Header <algorithm> synopsis**

```cpp
#include <initializer_list>
namespace std {
    namespace ranges {
        // 25.5, algorithm result types
        template<class I, class F>
        struct in_fun_result;
        template<class I1, class I2>
        struct in_in_result;
        template<class I, class O>
        struct in_out_result;
        template<class I1, class I2, class O>
        struct in_in_out_result;
        template<class I, class O1, class O2>
        struct in_out_out_result;
        template<class T>
        struct min_max_result;
        template<class I>
        struct in_found_result;
    }
    // 25.6, non-modifying sequence operations
    // 25.6.1, all of
    template<class InputIterator, class Predicate>
    constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred);
    template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    bool all_of(ExecutionPolicy&& exec, // see 25.3.5
        ForwardIterator first, ForwardIterator last, Predicate pred);
    namespace ranges {
        template<input_iterator I, sentinel_for<I> S, class Proj = identity,
            indirect_unary_predicate<projected<I, Proj>> Pred>
        constexpr bool all_of(I first, S last, Pred pred, Proj proj = {});
        template<input_range R, class Proj = identity,
            indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
        constexpr bool all_of(R&& r, Pred pred, Proj proj = {});
    }
}
```
// 25.6.2, any of

```cpp
template<class InputIterator, class Predicate>
constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator first, ForwardIterator last, Predicate pred);
```

```cpp
namespace ranges {
    template<input_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
        constexpr bool any_of(I first, S last, Pred pred, Proj proj = {});
    template<input_range R, class Proj = identity,
              indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
        constexpr bool any_of(R&& r, Pred pred, Proj proj = {});
}
```

// 25.6.3, none of

```cpp
template<class InputIterator, class Predicate>
constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator first, ForwardIterator last, Predicate pred);
```

```cpp
namespace ranges {
    template<input_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
        constexpr bool none_of(I first, S last, Pred pred, Proj proj = {});
    template<input_range R, class Proj = identity,
              indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
        constexpr bool none_of(R&& r, Pred pred, Proj proj = {});
}
```

// 25.6.4, for each

```cpp
template<class InputIterator, class Function>
constexpr Function for_each(InputIterator first, InputIterator last, Function f);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Function>
ForwardIterator first, ForwardIterator last, Function f);
```

```cpp
namespace ranges {
    template<input_iterator I, class F>
        using for_each_result = in_fun_result<I, F>;
    template<input_iterator I, sentinel_for<I> S, class Proj = identity,
             indirectly_unary_invocable<projected<I, Proj>> Fun>
        for_each_result<I, Fun>
        for_each(I first, S last, Fun f, Proj proj = {});
    template<input_range R, class Proj = identity,
              indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
        for_each_result<borrowed_iterator_t<R>, Fun>
        for_each(R&& r, Fun f, Proj proj = {});
}
```

```cpp
template<class InputIterator, class Size, class Function>
constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
ForwardIterator first, ForwardIterator last, Function f);
```

```cpp
namespace ranges {
    template<class I, class F>
        using for_each_n_result = in_fun_result<I, F>;
    template<class I, class F>
        using for_each_n_result = in_fun_result<I, F>;
}
```
template<input_iterator I, class Proj = identity,
    indirectly unary_invocable<projected<I, Proj>> Fun>
    constexpr for_each_n_result<I, Fun>
    for_each_n(I first, iter_difference_t<I> n, Fun f, Proj proj = {});

// 25.6.5, find
template<class InputIterator, class T>
    constexpr InputIterator find(InputIterator first, InputIterator last,
    const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
    ForwardIterator find(ExecutionPolicy&& exec, // see 25.3.5
    ForwardIterator first, ForwardIterator last,
    const T& value);

namespace ranges {
    template<input_iterator I, sentinel_for<I> S, class T, class Proj = identity>
        requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*> 
        constexpr I find(I first, S last, const T& value, Proj proj = {});
    template<input_range R, class T, class Proj = identity>
        requires indirect_binary_predicate<ranges::equal_to,
            projected<iterator_t<R>, Proj>, const T*> 
        constexpr borrowed_iterator_t<R>
        find(R&& r, const T& value, Proj proj = {});
    template<input_iterator I, sentinel_for<I> S, class Proj = identity,
        indirect_unary_predicate<projected<I, Proj>> Pred>
        constexpr I find_if(I first, S last, Pred pred, Proj proj = {});
    template<input_range R, class Proj = identity,
        indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
        constexpr borrowed_iterator_t<R>
        find_if(R&& r, Pred pred, Proj proj = {});
}

// 25.6.6, find end
template<class ForwardIterator1, class ForwardIterator2>
    constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
    constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_end(ExecutionPolicy&& exec, // see 25.3.5
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
find_end(ExecutionPolicy&& exec, // see 25.3.5
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);

namespace ranges {
    template<forward_iterator I1, sentinel_for<I1> S1, forward_iterator I2, sentinel_for<I2> S2,
             class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_comparable<I1, I2, Pred, Proj1, Proj2>
    constexpr subrange<I1>
    find_end(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
             Proj1 proj1 = {}, Proj2 proj2 = {});

    template<input_iterator I1, sentinel_for<I1> S1, forward_iterator I2, sentinel_for<I2> S2,
              class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_comparable<I1, I2, Pred, Proj1, Proj2>
    constexpr I1 find_first_of(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
                               Proj1 proj1 = {}, Proj2 proj2 = {});

    template<input_range R1, forward_range R2,
              class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_comparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr borrowed_subrange_t<R1>
    find_first_of(R1&& r1, R2&& r2, Pred pred = {},
                  Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.6.7, find first
template<class InputIterator, class ForwardIterator>
constexpr InputIterator
find_first_of(InputIterator first1, InputIterator last1,
              ForwardIterator first2, ForwardIterator last2);

template<class InputIterator, class ForwardIterator, class BinaryPredicate>
constexpr InputIterator
find_first_of(InputIterator first1, InputIterator last1,
              ForwardIterator first2, ForwardIterator last2,
              BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_first_of(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
find_first_of(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

namespace ranges {
    template<input_iterator I1, sentinel_for<I1> S1, forward_iterator I2, sentinel_for<I2> S2,
              class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_comparable<I1, I2, Pred, Proj1, Proj2>
    constexpr I1 find_first_of(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
                               Proj1 proj1 = {}, Proj2 proj2 = {});

    template<input_range R1, forward_range R2,
              class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_comparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr borrowed_iterator_t<R1>
    find_first_of(R1&& r1, R2&& r2, Pred pred = {},
                  Proj1 proj1 = {}, Proj2 proj2 = {});
}

§ 25.4
// 25.6.8, adjacent_find
template<class ForwardIterator>
constexpr ForwardIterator
adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
constexpr ForwardIterator
adjacent_find(ForwardIterator first, ForwardIterator last,
BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator
adjacent_find(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
ForwardIterator
adjacent_find(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last,
BinaryPredicate pred);

namespace ranges {
  template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
indirect_binary_predicate<projected<I, Proj>,
projected<I, Proj>> Pred = ranges::equal_to>
constexpr I adjacent_find(I first, S last, Pred pred = {},
Proj proj = {});

template<forward_range R, class Proj = identity,
indirect_binary_predicate<projected<iterator_t<R>, Proj>,
projected<iterator_t<R>, Proj>> Pred = ranges::equal_to>
constexpr borrowed_iterator_t<R>
adjacent_find(R&& r, Pred pred = {}, Proj proj = {});
}

// 25.6.9, count
template<class InputIterator, class T>
constexpr typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);

template<class ExecutionPolicy, class ForwardIterator, class T>
typename iterator_traits<ForwardIterator>::difference_type
count(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last, const T& value);

template<class InputIterator, class Predicate>
constexpr typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
typename iterator_traits<ForwardIterator>::difference_type
count_if(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last, Predicate pred);

namespace ranges {
  template<input_iterator I, sentinel_for<I> S, class T, class Proj = identity>
requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>
constexpr iter_difference_t<I>
count(I first, S last, const T& value, Proj proj = {});

template<input_range R, class T, class Proj = identity>
requires indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>,
const T*>
constexpr range_difference_t<R>
count(R&& r, const T& value, Proj proj = {});

  template<input_iterator I, sentinel_for<I> S, class Proj = identity,
indirect_unary_predicate<projected<I, Proj>> Pred>
constexpr iter_difference_t<I>
count_if(I first, S last, Pred pred, Proj proj = {});

template<input_range R, class Proj = identity,
indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
constexpr range_difference_t<R>
count_if(R&& r, Pred pred, Proj proj = {});
}
count_if(R&& r, Pred pred, Proj proj = {});
}

// 25.6.10, mismatch
template<class InputIterator1, class InputIterator2>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, BinaryPredicate pred);

namespace ranges {
    template<class I1, class I2>
    using mismatch_result = in_in_result<I1, I2>;

    template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_comparable<I1, I2, Pred, Proj1, Proj2>
    constexpr mismatch_result<I1, I2>
mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
Proj1 proj1 = {}, Proj2 proj2 = {});

    template<input_range R1, input_range R2,
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_comparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr mismatch_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>
mismatch(R1&& r1, R2&& r2, Pred pred = {},
Proj1 proj1 = {}, Proj2 proj2 = {});
}
// 25.6.11, equal
template<class InputIterator1, class InputIterator2>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2, BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
bool equal(ExecutionPolicy&& exec, // see 25.3.5
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
bool equal(ExecutionPolicy&& exec, // see 25.3.5
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, BinaryPredicate pred);

namespace ranges {
    template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
        class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
        requires indirectly_comparable<I1, I2, Pred, Proj1, Proj2>
    constexpr bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
        Pred pred = {},
        Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, input_range R2, class Pred = ranges::equal_to,
        class Proj1 = identity, class Proj2 = identity>
        requires indirectly_comparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr bool equal(R1&& r1, R2&& r2, Pred pred = {},
        Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.6.12, is permutation
template<class ForwardIterator1, class ForwardIterator2>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
namespace ranges {
    template<forward_iterator I1, sentinel_for<I1> S1, forward_iterator I2, sentinel_for<I2> S2, class Proj1 = identity, class Proj2 = identity, indirect_equivalence_relation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = ranges::equal_to>
    constexpr bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
        Pred pred = {},
        Proj1 proj1 = {}, Proj2 proj2 = {});  

    template<forward_range R1, forward_range R2, class Proj1 = identity, class Proj2 = identity, indirect_equivalence_relation<iterator_t<R1>, iterator_t<R2>, Proj1>, projected<iterator_t<R1>, Proj2>> Pred = ranges::equal_to>
    constexpr bool is_permutation(R1&& r1, R2&& r2, Pred pred = {},
        Proj1 proj1 = {}, Proj2 proj2 = {});  
}

// 25.6.13, search

namespace {
    template<class ForwardIterator1, class ForwardIterator2>
    constexpr ForwardIterator1
        search(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2, ForwardIterator2 last2);

    template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
    constexpr ForwardIterator1
        search(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2, ForwardIterator2 last2,
            BinaryPredicate pred);

    template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1
        search(ExecutionPolicy&& exec,
            ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2, ForwardIterator2 last2);

    template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
    ForwardIterator1
        search(ExecutionPolicy&& exec,
            ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2, ForwardIterator2 last2,
            BinaryPredicate pred);
}

namespace ranges {
    template<forward_iterator I1, sentinel_for<I1> S1, forward_iterator I2, sentinel_for<I2> S2, class Proj1 = identity, class Proj2 = identity, requires indirectly_comparable<I1, I2,Pred, Proj1, Proj2>
            subrange<I1>
        search(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
            Proj1 proj1 = {}, Proj2 proj2 = {});  

    template<forward_range R1, forward_range R2, class Proj1 = identity, class Proj2 = identity, requires indirectly_comparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
            borrowed subrange_t<R1>
        search(R1&& r1, R2&& r2, Pred pred = {},
            Proj1 proj1 = {}, Proj2 proj2 = {});  
}

template<class ForwardIterator, class Size, class T>
constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
        Size count, const T& value);

template<class ForwardIterator, class Size, class T, class BinaryPredicate>
constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
        Size count, const T& value, BinaryPredicate pred);

\section{25.4 1056}
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
ForwardIterator
search_n(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last,
Size count, const T& value);

template<class ExecutionPolicy, class ForwardIterator, class Size, class T,
class BinaryPredicate>
ForwardIterator
search_n(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last,
Size count, const T& value,
BinaryPredicate pred);

namespace ranges {
  template<forward_iterator I, sentinel_for<I> S, class T,
class Pred = ranges::equal_to, class Proj = identity>
requires indirectly_comparable<I, const T*, Pred, Proj>
constexpr subrange<I>
search_n(I first, S last, iter_difference_t<I> count,
const T& value, Pred pred = {}, Proj proj = {});
  template<forward_range R, class T, class Pred = ranges::equal_to,
class Proj = identity>
requires indirectly_comparable<iterator_t<R>, const T*, Pred, Proj>
constexpr borrowed_subrange_t<R>
search_n(R&& r, range_difference_t<R> count,
const T& value, Pred pred = {}, Proj proj = {});
}

template<class ForwardIterator, class Searcher>
constexpr ForwardIterator
search(ForwardIterator first, ForwardIterator last, const Searcher& searcher);

// 25.7, mutating sequence operations
// 25.7.1, copy
template<class InputIterator, class OutputIterator>
constexpr OutputIterator copy(InputIterator first, InputIterator last,
OutputIterator result);

namespace ranges {
  template<class I, class O>
  using copy_result = in_out_result<I, O>;

  template<input_iterator I, sentinel_for<I> S, weakly_incrementable O>
  requires indirectly_copyable<I, O>
  constexpr copy_result<I, O>
copy(I first, S last, O result);

  template<input_range R, weakly_incrementable O>
  requires indirectly_copyable<iterator_t<R>, O>
  constexpr copy_result<borrowed_iterator_t<R>, O>
copy(R&& r, O result);
}

template<class InputIterator, class Size, class OutputIterator>
constexpr OutputIterator copy_n(InputIterator first, Size n,
OutputIterator result);

namespace ranges {
  template<class I, class O>
  using copy_result = in_out_result<I, O>;

  template<input_iterator I, sentinel_for<I> S, weakly_incrementable O>
  requires indirectly_copyable<I, O>
  constexpr copy_result<I, O>
copy(I first, S last, O result);

  template<input_range R, weakly_incrementable O>
  requires indirectly_copyable<iterator_t<R>, O>
  constexpr copy_result<borrowed_iterator_t<R>, O>
copy(R&& r, O result);
}
namespace ranges {
  template<class I, class O>
  using copy_n_result = in_out_result<I, O>;

  template<input_iterator I, weakly_incrementable O>
  requires indirectly_copyable<I, O>
  constexpr copy_n_result<I, O>
  copy_n(I first, iter_difference_t<I> n, O result);
}

namespace ranges {
  template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
  OutputIterator result, Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
  class Predicate>
  ForwardIterator2 copy_if(ExecutionPolicy&& exec,
  ForwardIterator1 first, ForwardIterator1 last,
  ForwardIterator2 result, Predicate pred);
}

namespace ranges {
  template<class I1, class I2>
  using copy_backward_result = in_out_result<I1, I2>;

  template<bidirectional_iterator I1, sentinel_for<I1> S1, bidirectional_iterator I2>
  requires indirectly_copyable<I1, I2>
  constexpr copy_backward_result<I1, I2>
  copy_backward(I1 first, S1 last, I2 result);
}

namespace ranges {
  template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
  copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
  BidirectionalIterator2 result);
}

template<class InputIterator, class OutputIterator>
constexpr OutputIterator move(InputIterator first, InputIterator last,
  OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1,
  class ForwardIterator2>
ForwardIterator2 move(ExecutionPolicy&& exec,
  ForwardIterator1 first, ForwardIterator1 last,
  ForwardIterator2 result);
namespace ranges {
    template<class I, class O>
    using move_result = in_out_result<I, O>;

template<input_iterator I, sentinel_for<I> S, weakly_incrementable O>
    requires indirectly_movable<I, O>
    constexpr move_result<I, O>
    move(I first, S last, O result);

template<input_range R, weakly_incrementable O>
    requires indirectly_movable<iterator_t<R>, O>
    constexpr move_result<borrowed_iterator_t<R>, O>
    move(R&& r, O result);
}

namespace ranges {
    template<class I1, class I2>
    using move_backward_result = in_out_result<I1, I2>;

template<bidirectional_iterator I1, sentinel_for<I1> S1, bidirectional_iterator I2>
    requires indirectly_movable<I1, I2>
    constexpr move_backward_result<I1, I2>
    move_backward(I1 first, S1 last, I2 result);

template<bidirectional_range R, bidirectional_iterator I>
    requires indirectly_movable<iterator_t<R>, I>
    constexpr move_backward_result<borrowed_iterator_t<R>, I>
    move_backward(R&& r, I result);
}

// 25.7.3, swap
template<class ForwardIterator1, class ForwardIterator2>
constexpr ForwardIterator2
swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
swap_ranges(ExecutionPolicy&& exec,
            ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);

namespace ranges {
    template<class I1, class I2>
    using swap_ranges_result = in_in_result<I1, I2>;

template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2>
    requires indirectly_swappable<I1, I2>
    constexpr swap_ranges_result<I1, I2>
    swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

template<input_range R1, input_range R2>
    requires indirectly_swappable<iterator_t<R1>, iterator_t<R2>>
    constexpr swap_ranges_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>>
    swap_ranges(R1&& r1, R2&& r2);
}

template<class ForwardIterator1, class ForwardIterator2>
constexpr void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

// 25.7.4, transform
template<class InputIterator, class OutputIterator, class UnaryOperation>
constexpr OutputIterator
transform(InputIterator first1, InputIterator last1,
          OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator, 
    class BinaryOperation>
constexpr OutputIterator 
transform(InputIterator1 first1, InputIterator1 last1,  
        InputIterator2 first2, OutputIterator result, 
        BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, 
    class UnaryOperation>
ForwardIterator2 
transform(ExecutionPolicy && exec,  
        ForwardIterator1 first1, ForwardIterator1 last1, 
        ForwardIterator2 result, UnaryOperation op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, 
    class BinaryOperation>
ForwardIterator 
transform(ExecutionPolicy && exec,  
        ForwardIterator1 first1, ForwardIterator1 last1, 
        ForwardIterator2 first2, ForwardIterator result, 
        BinaryOperation binary_op);

namespace ranges {

template<class I, class O>
using unary_transform_result = in_out_result<I, O>;

template<input_iterator I, sentinel_for<I> S, weakly_incrementable 0, 
    copy_constructible F, class Proj = identity> 
requires indirectly_writable<O, indirect_result_t<F&>, projected<I, Proj>>
constexpr unary_transform_result<I, O> 
transform(I first1, S last1, O result, F op, Proj = {});

template<input_range R, weakly_incrementable 0, copy_constructible F, 
    class Proj = identity> 
requires indirectly_writable<O, indirect_result_t<F&>, projected<iterator_t<R>, Proj>>
constexpr unary_transform_result<borrowed_iterator_t<R>, O> 
transform(R&& r, O result, F op, Proj proj = {});

    template<class I1, class I2, class O>
using binary_transform_result = in_in_out_result<I1, I2, O>;

template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2, 
    weakly_incrementable 0, copy_constructible F, class Proj1 = identity, 
    class Proj2 = identity> 
requires indirectly_writable<O, indirect_result_t<F&>, projected<I1, Proj1>, 
    projected<I2, Proj2>>
constexpr binary_transform_result<I1, I2, O> 
transform(I1 first1, S1 last1, I2 first2, S2 last2, O result, 
        F binary_op, Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, input_range R2, weakly_incrementable 0, 
    copy_constructible F, class Proj1 = identity, class Proj2 = identity> 
requires indirectly_writable<O, indirect_result_t<F&>, projected<iterator_t<R1>, Proj1>, 
    projected<iterator_t<R2>, Proj2>>
constexpr binary_transform_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>, O> 
transform(R1&& r1, R2&& r2, O result, 
        F binary_op, Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.7.5, replace

template<class ForwardIterator, class T>
constexpr void replace(ForwardIterator first, ForwardIterator last, 
    const T& old_value, const T& new_value);

template<class ExecutionPolicy, class ForwardIterator, class T>
void replace(ExecutionPolicy && exec,  
    ForwardIterator first, ForwardIterator last, 
    const T& old_value, const T& new_value);
template<class ForwardIterator, class Predicate, class T>
constexpr void replace_if(ForwardIterator first, ForwardIterator last,
    Predicate pred, const T& new_value);

template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
void replace_if(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    Predicate pred, const T& new_value);

namespace ranges {
    template<input_iterator I, sentinel_for<I> S, class T1, class T2, class Proj = identity>
    requires indirectly_writable<I, const T2&> &&
        indirect_binary_predicate<ranges::equal_to, projected<I, Proj>>, const T1*> constexpr I
        replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = {});

    template<input_range R, class T1, class T2, class Proj = identity>
    requires indirectly_writable<iterator_t<R>, const T2&> &&
        indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>>, const T1*> constexpr I
        replace(R&& r, const T1& old_value, const T2& new_value, Proj proj = {});

    template<input_iterator I, sentinel_for<I> S, class T, class Proj = identity,
        indirect_unary_predicate<projected<I, Proj>> Pred>
    requires indirectly_writable<I, const T&> &&
        indirect_unary_predicate<projected<I, Proj>>, const T1*> constexpr I
        replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = {});

    template<input_range R, class T, class Proj = identity,
        indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
    requires indirectly_writable<iterator_t<R>, const T&> &&
        indirect_unary_predicate<projected<iterator_t<R>, Proj>>, const T1*> constexpr I
        replace_if(R&& r, Pred pred, const T& new_value, Proj proj = {});
}

namespace ranges {
    template<class InputIterator, class OutputIterator, class T>
    constexpr OutputIterator replace_copy(InputIterator first, InputIterator last,
        OutputIterator result,
        const T& old_value, const T& new_value);

    template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
    ForwardIterator2 replace_copy(ExecutionPolicy&& exec,
        ForwardIterator1 first, ForwardIterator1 last,
        ForwardIterator2 result,
        const T& old_value, const T& new_value);

    template<class InputIterator, class OutputIterator, class Predicate, class T>
    constexpr OutputIterator replace_copy_if(InputIterator first, InputIterator last,
        OutputIterator result,
        Predicate pred, const T& new_value);

    template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
        class Predicate, class T>
    ForwardIterator2 replace_copy_if(ExecutionPolicy&& exec,
        ForwardIterator1 first, ForwardIterator1 last,
        ForwardIterator2 result,
        Predicate pred, const T& new_value);
}

namespace ranges {
    template<class I, class O>
    using replace_copy_result = in_out_result<I, O>;

    template<input_iterator I, sentinel_for<I> S, class T1, class T2,
        output_iterator<const T2&> O, class Proj = identity>
    requires indirectly_copyable<I, O> &&
        indirect_binary_predicate<ranges::equal_to, projected<I, Proj>>, const T1*> constexpr I
    replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
        Proj proj = {});

§ 25.4 1061
template<input_range R, class T1, class T2, output_iterator<const T2&> O, class Proj = identity>
requires indirectly_copyable<iterator_t<R>, O> &&
indirect_binary_predicate<ranges::equal_to,
projected<iterator_t<R>, Proj>, const T1*>  
constexpr replace_copy_result<borrowed_iterator_t<R>, O>
replace_copy(R&& r, O result, const T1& old_value, const T2& new_value,
Proj proj = {});

template<class I, class O>
using replace_copy_if_result = in_out_result<I, O>;

template<input_iterator I, sentinel_for<I> S, class T, output_iterator<const T&> O, class Proj = identity, indirect_unary_predicate<projected<I, Proj>> Pred>
requires indirectly_copyable<I, O>  
constexpr replace_copy_if_result<I, O>
replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
Proj proj = {});

template<input_range R, class T, output_iterator<const T&> O, class Proj = identity, indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
requires indirectly_copyable<iterator_t<R>, O>  
constexpr replace_copy_if_result<borrowed_iterator_t<R>, O>
replace_copy_if(R&& r, O result, Pred pred, const T& new_value,
Proj proj = {});

// 25.7.6, fill
template<class ForwardIterator, class T>
constexpr void fill(ForwardIterator first, ForwardIterator last, const T& value);

template<class ExecutionPolicy, class ForwardIterator, class T>
void fill(ExecutionPolicy&& exec,  // see 25.3.5
ForwardIterator first, ForwardIterator last, const T& value);

namespace ranges {

template<class T, output_iterator<const T&> O, sentinel_for<O> S>
constexpr O fill(O first, S last, const T& value);

template<class T, output_range<const T&> R>
constexpr borrowed_iterator_t<R> fill(R&& r, const T& value);

template<class T, output_iterator<const T&> O>
constexpr O fill_n(O first, iter_difference_t<O> n, const T& value);
}

// 25.7.7, generate

namespace ranges {

template<class ForwardIterator, class Generator>
constexpr void generate(ForwardIterator first, ForwardIterator last, Generator gen);

template<class ExecutionPolicy, class ForwardIterator, class Generator>
void generate(ExecutionPolicy&& exec,  // see 25.3.5
ForwardIterator first, ForwardIterator last, Generator gen);

template<class OutputIterator, class Size, class Generator>
constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);

template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
ForwardIterator generate_n(ExecutionPolicy&& exec,  // see 25.3.5
ForwardIterator first, Size n, Generator gen);

§ 25.4
namespace ranges {
    template<input_or_output_iterator O, sentinel_for<O> S, copy_constructible F>
    requires invocable<F&> && indirectly_writable<O, invoke_result_t<F&>>
    constexpr O generate(O first, S last, F gen);
    template<class R, copy_constructible F>
    requires invocable<F&> && output_range<R, invoke_result_t<F&>>
    constexpr borrowed_iterator_t<R> generate(R&& r, F gen);
    template<input_or_output_iterator O, copy_constructible F>
    requires invocable<F&> && indirectly_writable<O, invoke_result_t<F&>>
    constexpr O generate_n(O first, iter_difference_t<O> n, F gen);
}

// 25.7.8, remove
template<class ForwardIterator, class T>
constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
ForwardIterator remove(ExecutionPolicy&& exec,
// see 25.3.5
ForwardIterator first, ForwardIterator last,
const T& value);

namespace ranges {
    template<permutable I, sentinel_for<I> S, class T, class Proj = identity>
    requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>.
    constexpr subrange<I> remove(I first, S last, const T& value, Proj proj = {});
    template<forward_range R, class T, class Proj = identity>
    requires permutable<iterator_t<R>> &&
    indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T>.
    constexpr borrowed_subrange_t<R>
    remove(R&& r, const T& value, Proj proj = {});
    template<permutable I, sentinel_for<I> S, class T = identity,
    indirect_unary_predicate<projected<I, Proj>>, Pred>
    constexpr subrange<I> remove_if(I first, S last, Pred pred, Proj proj = {});
    template<forward_range R, class Proj = identity,
    indirect_unary_predicate<projected<iterator_t<R>, Proj>>, Pred>
    requires permutable<iterator_t<R>>
    constexpr borrowed_subrange_t<R>
    remove_if(R&& r, Pred pred, Proj proj = {});
}

template<class InputIterator, class OutputIterator, class T>
constexpr OutputIterator
    remove_copy(InputIterator first, InputIterator last,
OutputIterator result, const T& value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class T>
ForwardIterator2
    remove_copy(ExecutionPolicy&& exec,
// see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, const T& value);

template<class InputIterator, class OutputIterator, class Predicate>
constexpr OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Predicate>
ForwardIterator2
remove_copy_if(ExecutionPolicy&& exec, // see 25.3.5
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 result, Predicate pred);

namespace ranges {
    template<class I, class O>
    using remove_copy_result = in_out_result<I, O>;

template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, class T, class Proj = identity>
    requires indirectly_copyable<I, O> &&
        indirect_binary_predicatranges::equal_to, projected<I, Proj>, const T>>
    constexpr remove_copy_result<I, O>
    remove_copy(I first, S last, O result, const T& value, Proj proj = {});

template<input_range R, weakly_incrementable O, class T, class Proj = identity>
    requires indirectly_copyable<iterator_t<R>, O> &&
        indirect_binary_predicatranges::equal_to,
        projected<iterator_t<R>, Proj>, const T>>
    constexpr remove_copy_result<borrowed_iterator_t<R>, O>
    remove_copy(R&& r, O result, const T& value, Proj proj = {});

    template<class I, class O>
    using remove_copy_if_result = in_out_result<I, O>;

template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, class Proj = identity, indirect_unary_predicate<projected<I, Proj>> Pred>
    requires indirectly_copyable<I, O>
    constexpr remove_copy_if_result<I, O>
    remove_copy_if(I first, S last, O result, Pred pred, Proj proj = {});

template<input_range R, weakly_incrementable O, class Proj = identity, indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
    requires indirectly_copyable<iterator_t<R>, O>
    constexpr remove_copy_if_result<borrowed_iterator_t<R>, O>
    remove_copy_if(R&& r, O result, Pred pred, Proj proj = {});
}

// 25.7.9, unique

template<class ForwardIterator>
constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last,
    BinaryPredicate pred);

namespace ranges {
    template<permutable I, sentinel_for<I> S, class Proj = identity, indirect_equivalence_relation<projected<I, Proj>> C = ranges::equal_to>
    constexpr subrange<I> unique(I first, S last, C comp = {}, Proj proj = {});

template<forward_range R, class Proj = identity, indirect_equivalence_relation<projected<iterator_t<R>, Proj>> C = ranges::equal_to>
    requires permutable<iterator_t<R>>
    constexpr borrowed_subrange_t<R>
    unique(R&& r, C comp = {}, Proj proj = {});
}
template<class InputIterator, class OutputIterator>
constexpr OutputIterator
unique_copy(InputIterator first, InputIterator last,
             OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
constexpr OutputIterator
unique_copy(InputIterator first, InputIterator last,
             OutputIterator result, BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
unique_copy(ExecutionPolicy&& exec, // see 25.3.5
            ForwardIterator1 first, ForwardIterator1 last,
            ForwardIterator2 result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator2
unique_copy(ExecutionPolicy&& exec, // see 25.3.5
            ForwardIterator1 first, ForwardIterator1 last,
            ForwardIterator2 result, BinaryPredicate pred);

namespace ranges {
    template<class I, class O>
    using unique_copy_result = in_out_result<I, O>;

    template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, class Proj = identity,
             indirect_equivalence_relation<projected<I>, Proj>> C = ranges::equal_to
    requires indirectly_copyable<I, O> &&
    (forward_iterator<I> || (input_iterator<O> && same_as<iter_value_t<I>, iter_value_t>O>) ||
      indirectly_copyable_storable<I, O>)
    constexpr unique_copy_result<I, O>
    unique_copy(I first, S last, O result, C comp = {}, Proj proj = {});

    template<input_range R, weakly_incrementable O, class Proj = identity,
             indirect_equivalence_relation<iterator_t<R>, Proj>> C = ranges::equal_to
    requires indirectly_copyable<iterator_t<R>, O> &&
    (forward_iterator<iterator_t<R>> || (input_iterator<O> && same_as<range_value_t<R>, iter_value_t>O>) ||
      indirectly_copyable_storable<iterator_t<R>, O>)
    constexpr unique_copy_result<borrowed_iterator_t<R>, O>
    unique_copy(R&& r, O result, C comp = {}, Proj proj = {});
}

// 25.7.10, reverse

template<class BidirectionalIterator>
constexpr void reverse(BidirectionalIterator first, BidirectionalIterator last);

template<class ExecutionPolicy, class BidirectionalIterator>
void reverse(ExecutionPolicy&& exec, // see 25.3.5
             BidirectionalIterator first, BidirectionalIterator last);

namespace ranges {
    template<bidirectional_iterator I, sentinel_for<I> S>
    requires permutable<I>
    constexpr I reverse(I first, S last);

    template<bidirectional_range R>
    requires permutable<iterator_t<R>>
    constexpr borrowed_iterator_t<R> reverse(R&& r);
}

template<class BidirectionalIterator, class OutputIterator>
constexpr OutputIterator
reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
             OutputIterator result);
template<class ExecutionPolicy, class BidirectionalIterator, class ForwardIterator>
ForwardIterator
reverse_copy(ExecutionPolicy&& exec, // see 25.3.5
           BidirectionalIterator first, BidirectionalIterator last,
           ForwardIterator result);

namespace ranges {
  template<class I, class O>
  using reverse_copy_result = in_out_result<I, O>;

  template<bidirectional_iterator I, sentinel_for<I> S, weakly_incrementable O>
  requires indirectly_copyable<I, O>
  constexpr reverse_copy_result<I, O>
  reverse_copy(I first, S last, O result);

  template<bidirectional_range R, weakly_incrementable O>
  requires indirectly_copyable<iterator_t<R>, O>
  constexpr reverse_copy_result<borrowed_iterator_t<R>, O>
  reverse_copy(R&& r, O result);
}

// 25.7.11, rotate
template<class ForwardIterator>
constexpr ForwardIterator rotate(ForwardIterator first,
                                 ForwardIterator middle,
                                 ForwardIterator last);

namespace ranges {
  template<permutable I, sentinel_for<I> S>
  constexpr subrange<I> rotate(I first, I middle, S last);

  template<forward_range R>
  requires permutable<iterator_t<R>>
  constexpr borrowed_subrange_t<R> rotate(R&& r, iterator_t<R> middle);
}

// 25.4 1066

template<class ForwardIterator, class OutputIterator>
constexpr OutputIterator
rotate_copy(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last, OutputIterator result);

namespace ranges {
  template<class I, class O>
  using rotate_copy_result = in_out_result<I, O>;

  template<forward_iterator I, sentinel_for<I> S, weakly_incrementable O>
  requires indirectly_copyable<I, O>
  constexpr rotate_copy_result<I, O>
  rotate_copy(I first, I middle, S last, O result);

  template<forward_range R, weakly_incrementable O>
  requires indirectly_copyable<iterator_t<R>, O>
  constexpr rotate_copy_result<borrowed_iterator_t<R>, O>
  rotate_copy(R&& r, iterator_t<R> middle, O result);
}
template<class PopulationIterator, class SampleIterator, class Distance, class UniformRandomBitGenerator>
SampleIterator sample(PopulationIterator first, PopulationIterator last, SampleIterator out, Distance n, UniformRandomBitGenerator&& g);

namespace ranges {
    template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, class Gen>
    requires (forward_iterator<I> || random_access_iterator<O>) &&
    indirectly_copyable<I, O> &&
    uniform_random_bit_generator<remove_reference_t<Gen>>
    O sample(I first, S last, O out, iter_difference_t<I> n, Gen&& g);
    template<input_range R, weakly_incrementable O, class Gen>
    requires (forward_range<R> || random_access_iterator<O>) &&
    indirectly_copyable<iterator_t<R>, O> &&
    uniform_random_bit_generator<remove_reference_t<Gen>>
    O sample(R&& r, O out, range_difference_t<R> n, Gen&& g);
}

namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Gen>
    requires permutable<I> &&
    uniform_random_bit_generator<remove_reference_t<Gen>>
    I shuffle(I first, S last, Gen&& g);
    template<random_access_range R, class Gen>
    requires permutable<iterator_t<R>> &&
    uniform_random_bit_generator<remove_reference_t<Gen>>
    borrowed_iterator_t<R> shuffle(R&& r, Gen&& g);
}

template<class RandomAccessIterator, class UniformRandomBitGenerator>
void shuffle(RandomAccessIterator first, RandomAccessIterator last, UniformRandomBitGenerator&& g);

namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Gen>
    requires permutable<I> &&
    uniform_random_bit_generator<remove_reference_t<Gen>>
    I shuffle(I first, S last, Gen&& g);
    template<random_access_range R, class Gen>
    requires permutable<iterator_t<R>> &&
    uniform_random_bit_generator<remove_reference_t<Gen>>
    borrowed_iterator_t<R> shuffle(R&& r, Gen&& g);
}

template<class ForwardIterator>
constexpr ForwardIterator
shift_left(ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator
shift_left(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);

template<class ForwardIterator>
constexpr ForwardIterator
shift_right(ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator
shift_right(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);
template<class RandomAccessIterator, class Compare>
constexpr void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator>
void sort(ExecutionPolicy&& exec, // see 25.3.5
          RandomAccessIterator first, RandomAccessIterator last);

namespace ranges {
  template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
           class Proj = identity>
constexpr I
  sort(I first, S last, Comp comp = {}, Proj proj = {});
}

namespace ranges {
  template<random_access_range R, class Comp = ranges::less, class Proj = identity>
constexpr borrowed_iterator_t<R>
  sort(R&& r, Comp comp = {}, Proj proj = {});
}

template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

namespace ranges {
  template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
           class Proj = identity>
  requires sortable<I, Comp, Proj>
  constexpr I
    stable_sort(I first, S last, Comp comp = {}, Proj proj = {});
}

namespace ranges {
  template<random_access_range R, class Comp = ranges::less, class Proj = identity>
  requires sortable<iterator_t<R>, Comp, Proj>
  constexpr borrowed_iterator_t<R>
    stable_sort(R&& r, Comp comp = {}, Proj proj = {});
}

template<class RandomAccessIterator, class Compare>
constexpr void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                             RandomAccessIterator last);

namespace ranges {
  template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
           class Proj = identity>
  requires sortable<I, Comp, Proj>
  constexpr void partial_sort(I first, I middle, I last, Comp comp);
}

namespace ranges {
  template<random_access_range R, class Comp = ranges::less, class Proj = identity>
  requires sortable<iterator_t<R>, Comp, Proj>
  constexpr void partial_sort(R&& r, I middle, I last, Comp comp);
}

§ 25.4 1068
namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less, 
             class Proj = identity>
    requires sortable<I, Comp, Proj>
    constexpr I
    partial_sort(I first, I middle, S last, Comp comp = {}, Proj proj = {});
    template<random_access_range R, class Comp = ranges::less, class Proj = identity>
    requires sortable<iterator_t<R>, Comp, Proj>
    constexpr borrowed_iterator_t<R>
    partial_sort(R&& r, iterator_t<R> middle, Comp comp = {}, Proj proj = {});
}

namespace ranges {
    template<class InputIterator, class RandomAccessIterator>
    constexpr RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last, 
                      RandomAccessIterator result_first, 
                      RandomAccessIterator result_last);
    template<class InputIterator, class RandomAccessIterator, class Compare>
    constexpr RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last, 
                      RandomAccessIterator result_first, 
                      RandomAccessIterator result_last, 
                      Compare comp);
    template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
    RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec, // see 25.3.5 
                      ForwardIterator first, ForwardIterator last, 
                      RandomAccessIterator result_first, 
                      RandomAccessIterator result_last);
    template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator, 
             class Compare>
    RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec, // see 25.3.5 
                      ForwardIterator first, ForwardIterator last, 
                      RandomAccessIterator result_first, 
                      RandomAccessIterator result_last, 
                      Compare comp);
}

namespace ranges {
    template<class I, class O>
    using partial_sort_copy_result = in_out_result<I, O>;
    template<input_iterator I1, sentinel_for<I1> S1, 
             random_access_iterator I2, sentinel_for<I2> S2, 
             class Comp = ranges::less, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_copyable<I1, I2> && sortable<I2, Comp, Proj2> && 
    indirect_strict_weak_order<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
    constexpr partial_sort_copy_result<I1, I2>
    partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last, 
                      Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
    template<input_range R1, random_access_range R2, class Comp = ranges::less, 
             class Proj1 = identity, class Proj2 = identity>
    requires indirectly_copyable<iterator_t<R1>, iterator_t<R2>> && 
    sortable<iterator_t<R2>, Comp, Proj2> && 
    indirect_strict_weak_order<Comp, projected<iterator_t<R1>, Proj1>, 
                             projected<iterator_t<R2>, Proj2>>
    constexpr partial_sort_copy_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>>
    partial_sort_copy(R1&& r, R2&& result_r, Comp comp = {}, 
                      Proj1 proj1 = {}, Proj2 proj2 = {});
}

template<class ForwardIterator>
constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
                        Compare comp);

template<class ExecutionPolicy, class ForwardIterator>
bool is_sorted(ExecutionPolicy&& exec, // see 25.3.5
               ForwardIterator first, ForwardIterator last);

namespace ranges {
    template<class ForwardIterator I, sentinel_for<I> S, class Proj = identity,
             indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
    constexpr bool is_sorted(I first, S last, Comp comp = {}, Proj proj = {});
    template<class ForwardRange R, class Proj = identity,
             indirect_strict_weak_order<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr bool is_sorted(R&& r, Comp comp = {}, Proj proj = {});
}

template<class ForwardIterator>
constexpr ForwardIterator
is_sorted_until(ForwardIterator first, ForwardIterator last);

namespace ranges {
    template<class ForwardIterator I, sentinel_for<I> S, class Proj = identity,
             indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
    constexpr I is_sorted_until(I first, S last, Comp comp = {}, Proj proj = {});
    template<class ForwardRange R, class Proj = identity,
             indirect_strict_weak_order<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr borrowed_iterator_t<R>
    is_sorted_until(R&& r, Comp comp = {}, Proj proj = {});
}

// 25.8.3, Nth element
template<class RandomAccessIterator>
constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                           RandomAccessIterator last);

namespace ranges {
    template<class RandomAccessIterator I, sentinel_for<I> S, class Proj = identity,
             indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
    constexpr void nth_element(I first, S last, Comp comp = {}, Proj proj = {});
    template<class ForwardRange R, class Proj = identity,
             indirect_strict_weak_order<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr void nth_element(R&& r, Comp comp = {}, Proj proj = {});
}

§ 25.4
namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
             class Proj = identity>
    requires sortable<I, Comp, Proj>
    constexpr I
    nth_element(I first, I nth, S last, Comp comp = {}, Proj proj = {});

template<random_access_range R, class Comp = ranges::less, class Proj = identity>
    requires sortable<iterator_t<R>, Comp, Proj>
    constexpr borrowed_iterator_t<R>
    nth_element(R&& r, iterator_t<R> nth, Comp comp = {}, Proj proj = {});
}

// 25.8.4, binary search
template<class ForwardIterator, class T>
    constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
    constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

namespace ranges {
    template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity,
             indirect_strict_weak_order<const T*, projected<I, Proj>> Comp = ranges::less>
    constexpr I lower_bound(I first, S last, const T& value, Comp comp = {}, Proj proj = {});

template<forward_range R, class T, class Proj = identity,
             indirect_strict_weak_order<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr borrowed_iterator_t<R>
    lower_bound(R&& r, const T& value, Comp comp = {}, Proj proj = {});
}

template<class ForwardIterator, class T>
    constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
    constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

namespace ranges {
    template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity,
             indirect_strict_weak_order<const T*, projected<I, Proj>> Comp = ranges::less>
    constexpr I upper_bound(I first, S last, const T& value, Comp comp = {}, Proj proj = {});

template<forward_range R, class T, class Proj = identity,
             indirect_strict_weak_order<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr borrowed_iterator_t<R>
    upper_bound(R&& r, const T& value, Comp comp = {}, Proj proj = {});
}

template<class ForwardIterator, class T>
    constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
    constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);
namespace ranges {
    template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity,
             indirect_strict_weak_order<const T*, projected<I, Proj>> Comp = ranges::less>
    constexpr subrange<I>
    equal_range(I first, S last, const T& value, Comp comp = {}, Proj proj = {});
    template<forward_range R, class T, class Proj = identity,
             indirect_strict_weak_order<const T*, projected<iterator_t<R>, Proj>> Comp =
             ranges::less>
    constexpr borrowed_subrange_t<R>
    equal_range(R&& r, const T& value, Comp comp = {}, Proj proj = {});
}

namespace ranges {
    template<class ForwardIterator, class T>
    constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value);
    template<class ForwardIterator, class T, class Compare>
    constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp);
}

namespace ranges {
    template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity,
             indirect_strict_weak_order<const T*, projected<I, Proj>> Comp = ranges::less>
    constexpr bool binary_search(I first, S last, const T& value, Comp comp = {},
                                  Proj proj = {});
    template<forward_range R, class T, class Proj = identity,
             indirect_strict_weak_order<const T*, projected<iterator_t<R>, Proj>> Comp =
             ranges::less>
    constexpr bool binary_search(R&& r, const T& value, Comp comp = {},
                                  Proj proj = {});
}

// 25.8.5, partitions
namespace ranges {
    template<input_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
    constexpr bool is_partitioned(I first, S last, Pred pred, Proj proj = {});
    template<input_range R, class Proj = identity,
             indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool is_partitioned(R&& r, Pred pred, Proj proj = {});
}

namespace ranges {
    template<input_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
    constexpr subrange<I>
    partition(I first, S last, Pred pred, Proj proj = {});
}

template<class ForwardIterator, class Predicate>
constexpr ForwardIterator partition(ForwardIterator first,
                                     ForwardIterator last,
                                     Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator partition(ExecutionPolicy&& exec, // see 25.3.5
                          ForwardIterator first,
                          ForwardIterator last,
                          Predicate pred);
template<forward_range R, class Proj = identity, 
  indirect_unary_predicate<Projected<iterator_t<R>>, Proj>> Pred>
requires permutable<iterator_t<R>>
constexpr borrowed_subrange_t<R>
    partition(R&& r, Pred pred, Proj proj = {});
}

template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first, 
    BidirectionalIterator last, Predicate pred);

namespace ranges {
    template<bidirectional_iterator I, sentinel_for<I> S, class Proj = identity, 
        indirect_unary_predicate<Projected<I>, Proj>> Pred>
    requires permutable<I>
    subrange<I> stable_partition(I first, S last, Pred pred, Proj proj = {});
    template<bidirectional_range R, class Proj = identity, 
        indirect_unary_predicate<iterator_t<R>, Proj>> Pred>
    requires permutable<iterator_t<R>>
    borrowed_subrange_t<R> stable_partition(R&& r, Pred pred, Proj proj = {});
}

namespace ranges {
    template<class InputIterator, class OutputIterator1, 
        class OutputIterator2, class Predicate>
    constexpr pair<OutputIterator1, OutputIterator2>
        partition_copy(InputIterator first, InputIterator last, 
        OutputIterator1 out_true, OutputIterator2 out_false, Predicate pred);
    template<class ExecutionPolicy, class ForwardIterator, class ForwardIterator1, 
        class ForwardIterator2, class Predicate>
    pair<ForwardIterator1, ForwardIterator2>
        partition_copy(ExecutionPolicy&& exec, 
        ForwardIterator first, ForwardIterator last, 
        ForwardIterator1 out_true, ForwardIterator2 out_false, Predicate pred);
}

namespace ranges {
    template<class I, class O1, class O2>
    using partition_copy_result = in_out_out_result<I, O1, O2>;
    template<input_iterator I, sentinel_for<I> S, 
        weakly_incrementable O1, weakly_incrementable O2, 
        class Proj = identity, indirect_unary_predicate<Projected<I>, Proj>> Pred>
    requires indirectly_copyable<I, O1> && indirectly_copyable<I, O2>
    constexpr partition_copy_result<I, O1, O2>
        partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred, 
        Proj proj = {});
    template<input_range R, weakly_incrementable O1, weakly_incrementable O2, 
        class Proj = identity, indirect_unary_predicate<Projected<iterator_t<R>>, Proj>> Pred>
    requires indirectly_copyable<iterator_t<R>, O1> && indirectly_copyable<iterator_t<R>, O2>
    constexpr partition_copy_result<borrowed_iterator_t<R>, O1, O2>
        partition_copy(R&& r, O1 out_true, O2 out_false, Pred pred, Proj proj = {});
}

§ 25.4 1073
template<class ForwardIterator, class Predicate>
constexpr ForwardIterator
partition_point(ForwardIterator first, ForwardIterator last,
               Predicate pred);

namespace ranges {
    template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
    constexpr I partition_point(I first, S last, Pred pred, Proj proj = {});
    template<forward_range R, class Proj = identity,
             indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr borrowed_iterator_t<R>
    partition_point(R&& r, Pred pred, Proj proj = {});
}

// 25.8.6, merge
template<class InputIterator1, class InputIterator2, class OutputIterator>
constexpr OutputIterator
merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator,
         class Compare>
constexpr OutputIterator
merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result, Compare comp);

namespace ranges {
    template<class I1, class I2, class O>
    using merge_result = in_in_out_result<I1, I2, O>;
    template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
              weakly_incrementable O, class Comp = ranges::less, class Proj1 = identity,
              class Proj2 = identity>
    requires mergeable<I1, I2, O, Comp, Proj1, Proj2>
    constexpr merge_result<I1, I2, O>
    merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
           Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
    template<input_range R1, input_range R2, weakly_incrementable O, class Comp = ranges::less,
              class Proj1 = identity, class Proj2 = identity>
    requires mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr merge_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>, O>
    merge(R1&& r1, R2&& r2, O result,
           Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
}
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
                 BidirectionalIterator middle,
                 BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
                 BidirectionalIterator middle,
                 BidirectionalIterator last, Compare comp);

namespace ranges {
  template<bidirectional_iterator I, sentinel_for<I> S, class Comp = ranges::less,
           class Proj = identity>
  requires sortable<I, Comp, Proj>
  I inplace_merge(I first, I middle, S last, Comp comp = {}, Proj proj = {});
  template<bidirectional_range R, class Comp = ranges::less, class Proj = identity>
  requires sortable<iterator_t<R>, Comp, Proj>
borrowed_iterator_t<R>
  inplace_merge(R&& r, iterator_t<R> middle, Comp comp = {},
                Proj proj = {});
}

// 25.8.7, set operations
namespace ranges {
  template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
           class Proj1 = identity, class Proj2 = identity,
           indirect_strict_weak_order<projected<I1, Proj1>, projected<I2, Proj2>> Comp =
ranges::less>
constexpr bool includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = {},
                        Proj1 proj1 = {}, Proj2 proj2 = {});
  template<input_range R1, input_range R2, class Proj1 = identity,
            class Proj2 = identity,
            indirect_strict_weak_order<projected<iterator_t<R1>, Proj1>,
            projected<iterator_t<R2>, Proj2>> Comp = ranges::less>
constexpr bool includes(R1&& r1, R2&& r2, Comp comp = {},
                        Proj1 proj1 = {}, Proj2 proj2 = {});
}
template<class InputIterator1, class InputIterator2, class OutputIterator>
constexpr OutputIterator
set_union(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
constexpr OutputIterator
set_union(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class Compare, class ForwardIterator>
ForwardIterator
set_intersection(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    ForwardIterator result);
namespace ranges {
    template<class I1, class I2, class O>
    using set_intersection_result = in_in_out_result<I1, I2, O>;

template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
    weakly_incrementable O, class Comp = ranges::less,
    class Proj1 = identity, class Proj2 = identity>
    constexpr set_intersection_result set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
    Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, input_range R2, weakly_incrementable O,
    class Comp = ranges::less, class Proj1 = identity, class Proj2 = identity>
    constexpr set_intersection_result set_intersection(borrowed_iterator_t<R1> first1, borrowed_iterator_t<R2> last2, O result,
    Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
set_difference(R1&& r1, R2&& r2, 0 result,
    Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
}

template<class InputIterator1, class InputIterator2, class OutputIterator>
constexpr OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
constexpr OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class ForwardIterator>
ForwardIterator
set_symmetric_difference(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    ForwardIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class Compare>
ForwardIterator
set_symmetric_difference(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    ForwardIterator result, Compare comp);

namespace ranges {
    template<class I1, class I2, class O>
    using set_symmetric_difference_result = in_in_out_result<I1, I2, O>;

    template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
        weakly_incrementable O, class Comp = ranges::less,
        class Proj1 = identity, class Proj2 = identity>
    requires mergeable<I1, I2, O, Comp, Proj1, Proj2>
   constexpr set_symmetric_difference_result<I1, I2, O>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
        Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

    template<input_range R1, input_range R2, weakly_incrementable O,
        class Comp = ranges::less, class Proj1 = identity, class Proj2 = identity>
    requires mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr set_symmetric_difference_result<borrowed_iterator_t<R1>,
        borrowed_iterator_t<R2>, O>
    set_symmetric_difference(R1&& r1, R2&& r2, 0 result, Comp comp = {},
        Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.8.8, heap operations

namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
        class Proj = identity>
    requires sortable<I, Comp, Proj>
    constexpr I
    push_heap(I first, S last, Comp comp = {}, Proj proj = {});
}

§ 25.4 1078
template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr borrowed_iterator_t<R>
push_heap(R&& r, Comp comp = {}, Proj proj = {});
}

template<class RandomAccessIterator>
constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less, class Proj = identity>
    requires sortable<I, Comp, Proj>
    constexpr I
    pop_heap(I first, S last, Comp comp = {}, Proj proj = {});
    template<random_access_range R, class Comp = ranges::less, class Proj = identity>
    constexpr borrowed_iterator_t<R>
pop_heap(R&& r, Comp comp = {}, Proj proj = {});
}

template<class RandomAccessIterator>
constexpr void make_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
constexpr void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less, class Proj = identity>
    requires sortable<I, Comp, Proj>
    constexpr I
    make_heap(I first, S last, Comp comp = {}, Proj proj = {});
    template<random_access_range R, class Comp = ranges::less, class Proj = identity>
    constexpr borrowed_iterator_t<R>
make_heap(R&& r, Comp comp = {}, Proj proj = {});
}

namespace ranges {
    template<class RandomAccessIterator>
    constexpr void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
constexpr void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

namespace ranges {
    template<class RandomAccess_iterator I, sentinel_for<I> S, class Comp = ranges::less, class Proj = identity>
    requires sortable<I, Comp, Proj>
    constexpr I
    sort_heap(I first, S last, Comp comp = {}, Proj proj = {});
    template<random_access_range R, class Comp = ranges::less, class Proj = identity>
    constexpr borrowed_iterator_t<R>
sort_heap(R&& r, Comp comp = {}, Proj proj = {});
}

template<class RandomAccessIterator>
constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator>
bool is_heap(ExecutionPolicy&& exec,
            // see 25.3.5
RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
bool is_heap(ExecutionPolicy&& exec,
            // see 25.3.5
RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);

namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Proj = identity,
            indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
    constexpr bool is_heap(I first, S last, Comp comp = {}, Proj proj = {});
    template<random_access_range R, class Proj = identity,
            indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr bool is_heap(R&& r, Comp comp = {}, Proj proj = {});
}

template<class RandomAccessIterator>
constexpr RandomAccessIterator
is_heap_until(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
constexpr RandomAccessIterator
is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator>
RandomAccessIterator
is_heap_until(ExecutionPolicy&& exec,
             // see 25.3.5
RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
RandomAccessIterator
is_heap_until(ExecutionPolicy&& exec,
             // see 25.3.5
RandomAccessIterator first, RandomAccessIterator last,
             Compare comp);

namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Proj = identity,
            indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
    constexpr I is_heap_until(I first, S last, Comp comp = {}, Proj proj = {});
    template<random_access_range R, class Proj = identity,
            indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr borrowed_iterator_t<R>
    is_heap_until(R&& r, Comp comp = {}, Proj proj = {});
}

// 25.8.9, minimum and maximum
template<class T> constexpr const T& min(const T& a, const T& b);
template<class T, class Compare>
constexpr const T& min(const T& a, const T& b, Compare comp);
template<class T>
constexpr T min(initializer_list<T> t);
template<class T, class Compare>
constexpr T min(initializer_list<T> t, Compare comp);

namespace ranges {
    template<class T, class Proj = identity,
            indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
    constexpr const T& min(const T& a, const T& b, Comp comp = {}, Proj proj = {});
    template<copyable T, class Proj = identity,
            indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
    constexpr T min(initializer_list<T> r, Comp comp = {}, Proj proj = {});
template<input_range R, class Proj = identity,  
    indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>  
requires indirectly_copyable_storable<iterator_t<R>, range_value_t<R>>  
constexpr range_value_t<R>  
    min(R&& r, Comp comp = {}, Proj proj = {});
}

template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>  
constexpr const T& max(const T& a, const T& b, Compare comp);
template<class T>  
constexpr T max(initializer_list<T> t);
template<class T, class Compare>  
constexpr T max(initializer_list<T> t, Compare comp);

namespace ranges {
    template<class T, class Proj = identity,  
        indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>  
    constexpr const T& max(const T& a, const T& b, Comp comp = {}, Proj proj = {});
    template<copyable T, class Proj = identity,  
        indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>  
    constexpr T max(initializer_list<T> r, Comp comp = {}, Proj proj = {});
    template<input_range R, class Proj = identity,  
        indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>  
requires indirectly_copyable_storable<iterator_t<R>, range_value_t<R>>  
constexpr range_value_t<R>  
    max(R&& r, Comp comp = {}, Proj proj = {});
}

namespace ranges {
    template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
    template<class T, class Compare>  
    constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
    template<class T>  
    constexpr pair<T, T> minmax(initializer_list<T> t);
    template<class T, class Compare>  
    constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);

    template<class T>  
    using minmax_result = min_max_result<T>;
    template<class T, class Proj = identity,  
        indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>  
    constexpr minmax_result<const T&>  
    minmax(const T& a, const T& b, Comp comp = {}, Proj proj = {});
    template<copyable T, class Proj = identity,  
        indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>  
    constexpr minmax_result<T>  
    minmax(initializer_list<T> r, Comp comp = {}, Proj proj = {});
    template<input_range R, class Proj = identity,  
        indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>  
requires indirectly_copyable_storable<iterator_t<R>, range_value_t<R>>  
constexpr minmax_result<range_value_t<R>>  
    minmax(R&& r, Comp comp = {}, Proj proj = {});
}

#include <ranges/algorithm/for_each.hpp>  

template<class ForwardIterator>  
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>  
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,  
    Compare comp);
template<class ExecutionPolicy, class ForwardIterator>  
ForwardIterator min_element(ExecutionPolicy&& exec,  
    ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator min_element(ExecutionPolicy&& exec, // see 25.3.5
FirstIterator, ForwardIterator last,
Compare comp);    // see 25.3.5
ForwardIterator first, ForwardIterator last,
namespace ranges {
    template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
        indirect_strict_weak_order<projected<I>, Proj>> Comp = ranges::less>
    constexpr I min_element(I first, S last, Comp comp = {}, Proj proj = {});
    template<forward_range R, class Proj = identity,
        indirect_strict_weak_order<projected<iterator_t<R>>, Proj>> Comp = ranges::less>
    constexpr borrowed_iterator_t<R>
    min_element(R&& r, Comp comp = {}, Proj proj = {});
}    // see 25.3.5
namespace ranges {
    template<class ForwardIterator>
    constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
    template<class ForwardIterator, class Compare>
    constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
Compare comp);    // see 25.3.5
    template<class ExecutionPolicy, class ForwardIterator>
    ForwardIterator max_element(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last);
    template<class ExecutionPolicy, class ForwardIterator, class Compare>
    ForwardIterator max_element(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last,
Compare comp);
}    // see 25.3.5
namespace ranges {
    template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
        indirect_strict_weak_order<projected<I>, Proj>> Comp = ranges::less>
    constexpr I max_element(I first, S last, Comp comp = {}, Proj proj = {});
    template<forward_range R, class Proj = identity,
        indirect_strict_weak_order<projected<iterator_t<R>>, Proj>> Comp = ranges::less>
    constexpr borrowed_iterator_t<R>
    max_element(R&& r, Comp comp = {}, Proj proj = {});
}    // see 25.3.5
namespace ranges {
    template<class ForwardIterator>
    constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
    template<class ForwardIterator, class Compare>
    constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
    template<class ExecutionPolicy, class ForwardIterator>
    pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last);
    template<class ExecutionPolicy, class ForwardIterator, class Compare>
    pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last, Compare comp);
}    // see 25.3.5
namespace ranges {
    template<class I>
    using minmax_element_result = min_max_result<I>;
    template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
        indirect_strict_weak_order<projected<I>, Proj>> Comp = ranges::less>
    constexpr pair<forward_iterator, forward_iterator>
    minmax_element_result(I first, S last, Comp comp = {}, Proj proj = {});
    template<forward_range R, class Proj = identity,
        indirect_strict_weak_order<projected<iterator_t<R>>, Proj>> Comp = ranges::less>
    constexpr pair<forward_iterator, forward_iterator>
    minmax_element_result(borrowed_iterator_t<R> r, Comp comp = {}, Proj proj = {});
}    // see 25.3.5
minmax_element(R&& r, Comp comp = {}, Proj proj = {});
}

// 25.8.10, bounded value
template<class T>
constexpr const T& clamp(const T& v, const T& lo, const T& hi);
template<class T, class Compare>
constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp);

namespace ranges {
    template<class T, class Proj = identity,
             indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
    constexpr const T&
    clamp(const T& v, const T& lo, const T& hi, Comp comp = {}, Proj proj = {});
}

// 25.8.11, lexicographical comparison
template<class InputIterator1, class InputIterator2>
constexpr bool
lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
constexpr bool
lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
Compare comp);

namespace ranges {
    template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
             class Proj1 = identity, class Proj2 = identity,
             indirect_strict_weak_order<projected<I1, Proj1>, projected<I2, Proj2>> Comp =
ranges::less>
    constexpr bool
    lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
    template<input_range R1, input_range R2, class Proj1 = identity,
             class Proj2 = identity,
             indirect_strict_weak_order<projected<iterator_t<R1>, Proj1>,
projected<iterator_t<R2>, Proj2>> Comp = ranges::less>
    constexpr bool
    lexicographical_compare(R1&& r1, R2&& r2, Comp comp = {},
Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.8.12, three-way comparison algorithms
template<class InputIterator1, class InputIterator2, class Cmp>
constexpr auto
lexicographical_compare_three_way(InputIterator1 b1, InputIterator1 e1,
InputIterator2 b2, InputIterator2 e2,
Cmp comp)
-> decltype(comp(*b1, *b2));
template<
class InputIterator1, class InputIterator2>
constexpr auto
lexicographical_compare_three_way(InputIterator1 b1, InputIterator1 e1,
InputIterator2 b2, InputIterator2 e2);

// 25.8.13, permutations

template<class BidirectionalIterator>
constexpr bool next_permutation(BidirectionalIterator first,
BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
constexpr bool next_permutation(BidirectionalIterator first,
BidirectionalIterator last, Compare comp);

namespace ranges {

template<class I>
using next_permutation_result = in_found_result<I>;

template<
bidirectional_iterator I, sentinel_for<I> S, class Comp = ranges::less,
class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr next_permutation_result<I>
next_permutation(I first, S last, Comp comp = {}, Proj proj = {});

template<
bidirectional_range R, class Comp = ranges::less,
class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr next_permutation_result<borrowed_iterator_t<R>>
next_permutation(R&& r, Comp comp = {}, Proj proj = {});
}

namespace ranges {

template<class BidirectionalIterator>
constexpr bool prev_permutation(BidirectionalIterator first,
BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
constexpr bool prev_permutation(BidirectionalIterator first,
BidirectionalIterator last, Compare comp);

namespace ranges {

template<class I>
using prev_permutation_result = in_found_result<I>;

template<
bidirectional_iterator I, sentinel_for<I> S, class Comp = ranges::less,
class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr prev_permutation_result<I>
prev_permutation(I first, S last, Comp comp = {}, Proj proj = {});

template<
bidirectional_range R, class Comp = ranges::less,
class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr prev_permutation_result<borrowed_iterator_t<R>>
prev_permutation(R&& r, Comp comp = {}, Proj proj = {});
}

25.5 Algorithm result types

Each of the class templates specified in this subclause has the template parameters, data members, and
special members specified below, and has no base classes or members other than those specified.

namespace std::ranges {

template<class I, class F>
struct in_fun_result {
 [[no_unique_address]] I in;
 [[no_unique_address]] F fun;
}
template<class I2, class F2>
    requires convertible_to<const I&, I2> && convertible_to<F, F2>
constexpr operator in_fun_result<I2, F2>() const & {
    return {in, fun};
}

template<class I2, class F2>
    requires convertible_to<I, I2> && convertible_to<F, F2>
constexpr operator in_fun_result<I2, F2>() && {
    return {std::move(in), std::move(fun)};
}
};

template<class I1, class I2>
struct in_in_result {
    [[no_unique_address]] I1 in1;
    [[no_unique_address]] I2 in2;

    template<class II1, class II2>
        requires convertible_to<const I1&, II1> && convertible_to<const I2&, II2>
    constexpr operator in_in_result<II1, II2>() const & {
        return {in1, in2};
    }

    template<class II1, class II2>
        requires convertible_to<I1, II1> && convertible_to<I2, II2>
    constexpr operator in_in_result<II1, II2>() && {
        return {std::move(in1), std::move(in2)};
    }
};

template<class I, class O>
struct in_out_result {
    [[no_unique_address]] I in;
    [[no_unique_address]] O out;

    template<class I2, class O2>
        requires convertible_to<const I&, I2> && convertible_to<const O&, O2>
    constexpr operator in_out_result<I2, O2>() const & {
        return {in, out};
    }

    template<class I2, class O2>
        requires convertible_to<I, I2> && convertible_to<O, O2>
    constexpr operator in_out_result<I2, O2>() && {
        return {std::move(in), std::move(out)};
    }
};

template<class I1, class I2, class O>
struct in_in_out_result {
    [[no_unique_address]] I1 in1;
    [[no_unique_address]] I2 in2;
    [[no_unique_address]] O out;

    template<class II1, class II2, class OO>
        requires convertible_to<const I1&, II1> && convertible_to<const I2&, II2> && convertible_to<const O&, OO>
    constexpr operator in_in_out_result<II1, II2, OO>() const & {
        return {in1, in2, out};
    }
};
template<class II1, class II2, class OO>
requires convertible_to<II1, II1> &&
convertible_to<II2, II2> &&
convertible_to<OO, OO>
constexpr operator in_in_out_result<II1, II2, OO>() && {
  return {std::move(in1), std::move(in2), std::move(out)};
}
};

template<class I, class O1, class O2>
struct in_out_out_result {
  [[no_unique_address]] I in;
  [[no_unique_address]] O1 out1;
  [[no_unique_address]] O2 out2;

  template<class II, class OO1, class OO2>
  requires convertible_to<const I&, II> &&
  convertible_to<const O1&, OO1> &&
  convertible_to<const O2&, OO2>
  constexpr operator in_out_out_result<II, OO1, OO2>() const & {
    return {in, out1, out2};
  }
  template<class II, class OO1, class OO2>
  requires convertible_to<I, II> &&
  convertible_to<O1, OO1> &&
  convertible_to<O2, OO2>
  constexpr operator in_out_out_result<II, OO1, OO2>() && {
    return {std::move(in), std::move(out1), std::move(out2)};
  }
};

template<class T>
struct min_max_result {
  [[no_unique_address]] T min;
  [[no_unique_address]] T max;

  template<class T2>
  requires convertible_to<const T&, T2>
  constexpr operator min_max_result<T2>() const & {
    return {min, max};
  }
  template<class T2>
  requires convertible_to<T, T2>
  constexpr operator min_max_result<T2>() && {
    return {std::move(min), std::move(max)};
  }
};

template<class I>
struct in_found_result {
  [[no_unique_address]] I in;
  bool found;

  template<class I2>
  requires convertible_to<const I&, I2>
  constexpr operator in_found_result<I2>() const & {
    return {in, found};
  }
  template<class I2>
  requires convertible_to<I, I2>
  constexpr operator in_found_result<I2>() && {
    return {std::move(in), found};
}
25.6 Non-modifying sequence operations

25.6.1 All of

```cpp
template<class InputIterator, class Predicate>
constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool all_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
            Predicate pred);
```

```cpp
template<input_iterator I, sentinel_for<I> S, class Proj = identity,
        indirect_unary_predicate<projected<I, Proj>> Pred>
constexpr bool ranges::all_of(I first, S last, Pred pred, Proj proj = {});
```

```cpp
template<input_range R, class Proj = identity,
        indirect_unary_predicate<iterator_t<R>, Proj>> Pred>
constexpr bool ranges::all_of(R&& r, Pred pred, Proj proj = {});
```

Let \( E \) be:

1. \( \text{pred}(\ast i) \) for the overloads in namespace \texttt{std};
2. \( \text{invoke(pred, invoke(proj, \ast i))} \) for the overloads in namespace \texttt{ranges}.

\textbf{Returns:} false if \( E \) is false for some iterator \( i \) in the range \([\text{first, last})\), and true otherwise.

\textbf{Complexity:} At most \( \text{last - first} \) applications of the predicate and any projection.

25.6.2 Any of

```cpp
template<class InputIterator, class Predicate>
constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool any_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
            Predicate pred);
```

```cpp
template<input_iterator I, sentinel_for<I> S, class Proj = identity,
        indirect_unary_predicate<projected<I, Proj>> Pred>
constexpr bool ranges::any_of(I first, S last, Pred pred, Proj proj = {});
```

```cpp
template<input_range R, class Proj = identity,
        indirect_unary_predicate<iterator_t<R>, Proj>> Pred>
constexpr bool ranges::any_of(R&& r, Pred pred, Proj proj = {});
```

Let \( E \) be:

1. \( \text{pred}(\ast i) \) for the overloads in namespace \texttt{std};
2. \( \text{invoke(pred, invoke(proj, \ast i))} \) for the overloads in namespace \texttt{ranges}.

\textbf{Returns:} true if \( E \) is true for some iterator \( i \) in the range \([\text{first, last})\), and false otherwise.

\textbf{Complexity:} At most \( \text{last - first} \) applications of the predicate and any projection.

25.6.3 None of

```cpp
template<class InputIterator, class Predicate>
constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool none_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
             Predicate pred);
```

```cpp
template<input_iterator I, sentinel_for<I> S, class Proj = identity,
        indirect_unary_predicate<projected<I, Proj>> Pred>
constexpr bool ranges::none_of(I first, S last, Pred pred, Proj proj = {});
```

\textbf{§ 25.6.3 1087}
template<input_range R, class Proj = identity,
indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
constexpr bool ranges::none_of(R&& r, Pred pred, Proj proj = {});

Let \( E \) be:

1. \( \text{pred}(\ast i) \) for the overloads in namespace \texttt{std};
2. \( \text{invoke(pred, invoke(proj, \ast i))} \) for the overloads in namespace \texttt{ranges}.

**Returns:** \text{false} if \( E \) is true for some iterator \( i \) in the range \([\text{first}, \text{last})\), and \text{true} otherwise.

**Complexity:** At most \( \text{last} - \text{first} \) applications of the predicate and any projection.

### 25.6.4 For each

#### [alg.foreach]

```cpp
template<class InputIterator, class Function>
constexpr Function for_each(InputIterator first, InputIterator last, Function f);
```

**Preconditions:** \texttt{Function} meets the \textit{Cpp17MoveConstructible} requirements (Table 28).

**Note 1:** \texttt{Function} need not meet the requirements of \textit{Cpp17CopyConstructible} (Table 29). — end note

**Effects:** Applies \( f \) to the result of dereferencing every iterator in the range \([\text{first}, \text{last})\), starting from \text{first} and proceeding to \text{last} - 1.

**Note 2:** If the type of \texttt{first} meets the requirements of a mutable iterator, \( f \) can apply non-constant functions through the dereferenced iterator. — end note

**Returns:** \( f \).

**Complexity:** Applies \( f \) exactly \( \text{last} - \text{first} \) times.

**Remarks:** If \( f \) returns a result, the result is ignored.

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Function>
void for_each(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last, Function f);
```

**Preconditions:** \texttt{Function} meets the \textit{Cpp17CopyConstructible} requirements.

**Effects:** Applies \( f \) to the result of dereferencing every iterator in the range \([\text{first}, \text{last})\).

**Note 3:** If the type of \texttt{first} meets the requirements of a mutable iterator, \( f \) can apply non-constant functions through the dereferenced iterator. — end note

**Complexity:** Applies \( f \) exactly \( \text{last} - \text{first} \) times.

**Remarks:** If \( f \) returns a result, the result is ignored. Implementations do not have the freedom granted under 25.3.3 to make arbitrary copies of elements from the input sequence.

**Note 4:** Does not return a copy of its \texttt{Function} parameter, since parallelization often does not permit efficient state accumulation. — end note

```cpp
template<input_iterator I, sentinel_for<I> S, class Proj = identity,
indirectly_unary_invocable<projected<I, Proj>> Fun>
constexpr ranges::for_each_result<I, Fun> ranges::for_each(I first, S last, Fun f, Proj proj = {});
```

**Effects:** Calls \( \text{invoke(f, invoke(proj, \ast i))} \) for every iterator \( i \) in the range \([\text{first}, \text{last})\), starting from \text{first} and proceeding to \text{last} - 1.

**Note 5:** If the result of \( \text{invoke(proj, \ast i)} \) is a mutable reference, \( f \) can apply non-constant functions. — end note

**Returns:** \( \{\text{last, std::move(f)}\} \).

**Complexity:** Applies \( f \) and \( \text{proj} \) exactly \( \text{last} - \text{first} \) times.

**Remarks:** If \( f \) returns a result, the result is ignored.

**Note 6:** The overloads in namespace \texttt{ranges} require \texttt{Fun} to model \textit{copy_constructible}. — end note

§ 25.6.4
template<class InputIterator, class Size, class Function>
constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);

Mandates: The type Size is convertible to an integral type (7.3.9, 11.4.8).
Preconditions: n >= 0 is true. Function meets the Cpp17MoveConstructible requirements.
[Note 7: Function need not meet the requirements of Cpp17CopyConstructible. — end note]
Effects: Applies f to the result of dereferencing every iterator in the range \([\text{first}, \text{first} + n)\) in order.
[Note 8: If the type of first meets the requirements of a mutable iterator, f can apply non-constant functions through the dereferenced iterator. — end note]

Returns: first + n.
Remarks: If f returns a result, the result is ignored.

template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
ForwardIterator for_each_n(ExecutionPolicy&& exec, ForwardIterator first, Size n, Function f);

Mandates: The type Size is convertible to an integral type (7.3.9, 11.4.8).
Preconditions: n >= 0 is true. Function meets the Cpp17CopyConstructible requirements.
Effects: Applies f to the result of dereferencing every iterator in the range \([\text{first}, \text{first} + n)\).
[Note 9: If the type of first meets the requirements of a mutable iterator, f can apply non-constant functions through the dereferenced iterator. — end note]

Returns: first + n.
Remarks: If f returns a result, the result is ignored. Implementations do not have the freedom granted under 25.3.3 to make arbitrary copies of elements from the input sequence.

template<input_iterator I, class Proj = identity, indirectly_unary_invocable<projected<I, Proj>> Fun>
ranges::for_each_n_result<I, Fun>
ranges::for_each_n(I first, iter_difference_t<I> n, Fun f, Proj proj = {});

Preconditions: n >= 0 is true.
Effects: Calls invoke(f, invoke(proj, *i)) for every iterator i in the range \([\text{first}, \text{first} + n)\) in order.
[Note 10: If the result of invoke(proj, *i) is a mutable reference, f can apply non-constant functions. — end note]

Returns: \{first + n, std::move(f)\}.
Remarks: If f returns a result, the result is ignored.
[Note 11: The overload in namespace ranges requires Fun to model copy_constructible. — end note]

25.6.5 Find

[alg.find]

template<class InputIterator, class T>
constexpr InputIterator find(InputIterator first, InputIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
ForwardIterator find(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, const T& value);

template<class InputIterator, class Predicate>
constexpr InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator find_if(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, Predicate pred);

template<class InputIterator, class Predicate>
constexpr InputIterator find_if_not(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator find_if_not(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last,
Predicate pred);

template<input_iterator I, sentinel_for<I> S, class T, class Proj = identity>
requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>
constexpr I ranges::find(I first, S last, const T& value, Proj proj = {});

template<input_range R, class T, class Proj = identity>
requires indirect_unary_predicate<projected<iterator_t<R>, Proj>, const T*>
constexpr borrowed_iterator_t<R> ranges::find_if_not(R&& r, const T& value, Proj proj = {});

Let \( E \) be:

1. \( *i == \text{value} \) for find;
2. \( \text{pred}(i) \neq \text{false} \) for find_if;
3. \( \text{pred}(i) == \text{false} \) for find_if_not;
4. \( \text{bool}(\text{invoke(proj, } i\text{)} == \text{value}) \) for ranges::find;
5. \( \text{bool}(\text{involve(pred, invoke(proj, } i\text{)}) for ranges::find_if;
6. \( \text{bool}(!\text{involve(pred, invoke(proj, } i\text{)}) for ranges::find_if_not.

Returns: The first iterator \( i \) in the range \([first, last)\) for which \( E \) is true. Returns last if no such iterator is found.

Complexity: At most last \(-\) first applications of the corresponding predicate and any projection.

25.6.6 Find end

template<class ForwardIterator1, class ForwardIterator2>
constexpr ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
constexpr ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);

§ 25.6.6 1090
ForwardIterator2 first2, ForwardIterator2 last2,
    BinaryPredicate pred);

template<forward_iterator I1, sentinel_for<I1> S1, forward_iterator I2, sentinel_for<I2> S2,
    class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_comparable<I1, I2, Pred, Proj1, Proj2>
    constexpr subrange<I1>
    ranges::find_end(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
        Proj1 proj1 = {}, Proj2 proj2 = {});

template<forward_range R1, forward_range R2,
    class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires indirectly_comparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr borrowed_subrange_t<R1>
    ranges::find_end(R1&& r1, R2&& r2, Pred pred = {},
        Proj1 proj1 = {}, Proj2 proj2 = {});

Let:

— (1.1) pred be equal_to{} for the overloads with no parameter pred;
— (1.2) E be:
  — (1.2.1) pred(*(i + n), *(first2 + n)) for the overloads in namespace std;
  — (1.2.2) invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n))) for the overloads in
  namespace ranges;
— (1.3) i be last1 if (first2, last2) is empty, or if (last2 - first2) > (last1 - first1) is true,
  or if there is no iterator in the range [first1, last1 - (last2 - first2)) such that for every
  non-negative integer n < (last2 - first2), E is true. Otherwise i is the last such iterator in
  [first1, last1 - (last2 - first2)).

Returns:

— i for the overloads in namespace std.
— {i, i + (i == last1 ? 0 : last2 - first2)) for the overloads in namespace ranges.

Complexity: At most (last2 - first2) * (last1 - first1 - (last2 - first2) + 1) applications
of the corresponding predicate and any projections.

25.6.7 Find first

template<class InputIterator, class ForwardIterator>
    constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
        ForwardIterator first2, ForwardIterator last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1
    find_first_of(ExecutionPolicy&& exec,
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator, class ForwardIterator,
    class BinaryPredicate>
    constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
        ForwardIterator first2, ForwardIterator last2,
        BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
    ForwardIterator1
    find_first_of(ExecutionPolicy&& exec,
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
template<input_iterator I1, sentinel_for<I1> S1, forward_iterator I2, sentinel_for<I2> S2, 
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity> 
requires indirectly_comparable<I1, I2, Pred, Proj1, Proj2> 
constexpr I1 ranges::find_first_of(I1 first1, S1 last1, I2 first2, S2 last2, 
Pred pred = {}, 
Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, forward_range R2, 
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity> 
requires indirectly_comparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2> 
constexpr borrowed_iterator_t<R1> 
ranges::find_first_of(R1&& r1, R2&& r2, 
Pred pred = {}, 
Proj1 proj1 = {}, Proj2 proj2 = {});

Let $E$ be:

1. $(1.1)$ \[ i == j \] for the overloads with no parameter $pred$;
2. $(1.2)$ \[ pred(*i, *j) != false \] for the overloads with a parameter $pred$ and no parameter $proj$;
3. $(1.3)$ \[ bool(invoke(pred, invoke(proj1, *i), invoke(proj2, *j))) \] for the overloads with parameters $pred$ and $proj$.

Effects: Finds an element that matches one of a set of values.

Returns: The first iterator $i$ in the range $[first1, last1)$ such that for some iterator $j$ in the range $[first2, last2)$ $E$ holds. Returns $last1$ if $[first2, last2)$ is empty or if no such iterator is found.

Complexity: At most $(last1-first1) * (last2-first2)$ applications of the corresponding predicate and any projections.

25.6.8 Adjacent find

template<class ForwardIterator> 
constexpr ForwardIterator 
adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator> 
ForwardIterator 
adjacent_find(ExecutionPolicy&& exec, 
ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate> 
constexpr ForwardIterator 
adjacent_find(ForwardIterator first, ForwardIterator last, 
BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate> 
ForwardIterator 
adjacent_find(ExecutionPolicy&& exec, 
ForwardIterator first, ForwardIterator last, 
BinaryPredicate pred);

template<forward_iterator I, sentinel_for<I> S, class Proj = identity, 
indirect_binary_predicate<projected<I, Proj>, 
projected<I, Proj>> Pred = ranges::equal_to> 
constexpr I ranges::adjacent_find(I first, S last, Pred pred = {}, Proj proj = {});

template<formal_range R, class Proj = identity, 
indirect_binary_predicate<projected<iterator_t<R>, Proj>, 
projected<iterator_t<R>, Proj>> Pred = ranges::equal_to> 
constexpr borrowed_iterator_t<R> ranges::adjacent_find(R&& r, Pred pred = {}, Proj proj = {});

Let $E$ be:

1. $(1.1)$ \[ i == *(i + 1) \] for the overloads with no parameter $pred$;
2. $(1.2)$ \[ pred(*i, *(i + 1)) != false \] for the overloads with a parameter $pred$ and no parameter $proj$;
3. $(1.3)$ \[ bool(invoke(pred, invoke(proj, *i), invoke(proj, *(i + 1)))) \] for the overloads with both parameters $pred$ and $proj$. 

§ 25.6.8
Returns: The first iterator \( i \) such that both \( i \) and \( i + 1 \) are in the range \([\text{first}, \text{last})\) for which \( E \) holds. Returns \( \text{last} \) if no such iterator is found.

Complexity: For the overloads with no ExecutionPolicy, exactly

\[
\min((i - \text{first}) + 1, (\text{last} - \text{first}) - 1)
\]

applications of the corresponding predicate, where \( i \) is adjacent_find's return value. For the overloads with an ExecutionPolicy, \( O(\text{last} - \text{first}) \) applications of the corresponding predicate, and no more than twice as many applications of any projection.

### 25.6.9 Count

[alg.count]

```cpp
template<class InputIterator, class T>
constexpr typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);

template<class ExecutionPolicy, class ForwardIterator, class T>
constexpr typename iterator_traits<ForwardIterator>::difference_type
count(ExecutionPolicy&& exec,
     ForwardIterator first, ForwardIterator last, const T& value);
```

```cpp
template<class InputIterator, class Predicate>
constexpr typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
constexpr typename iterator_traits<ForwardIterator>::difference_type
count_if(ExecutionPolicy&& exec,
         ForwardIterator first, ForwardIterator last, Predicate pred);
```

Let \( E \) be:

(1.1) \( \ast i = \text{value} \) for the overloads with no parameter \( \text{pred} \) or \( \text{proj} \);

(1.2) \( \text{pred}(\ast i) \neq \text{false} \) for the overloads with a parameter \( \text{pred} \) but no parameter \( \text{proj} \);

(1.3) \( \text{invoke}(\text{proj}, \ast i) = \text{value} \) for the overloads with a parameter \( \text{proj} \) but no parameter \( \text{pred} \);

(1.4) \( \text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i))) \) for the overloads with both parameters \( \text{proj} \) and \( \text{pred} \).

Effects: Returns the number of iterators \( i \) in the range \([\text{first}, \text{last})\) for which \( E \) holds.

Complexity: Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate and any projection.

### 25.6.10 Mismatch

[mismatch]

```cpp
template<class InputIterator1, class InputIterator2>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2);
```

§ 25.6.10
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
pair<ForwardIterator1, ForwardIterator2>
mismatch(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2);

template<class InputIterator1, class InputIterator2,
class BinaryPredicate>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
pair<ForwardIterator1, ForwardIterator2>
mismatch(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2,
class BinaryPredicate>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2, BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
pair<ForwardIterator1, ForwardIterator2>
mismatch(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2,
class BinaryPredicate>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2, BinaryPredicate pred);

template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
requires indirectly_comparable<I1, I2, Pred, Proj1, Proj2>
constexpr ranges::mismatch_result<I1, I2>
ranges::mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
    Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, input_range R2,
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
requires indirectly_comparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
constexpr ranges::mismatch_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>>
ranges::mismatch(R1&& r1, R2&& r2, Pred pred = {},
    Proj1 proj1 = {}, Proj2 proj2 = {});

Let last2 be first2 + (last1 - first1) for the overloads with no parameter last2 or r2.

Let \( E \) be:

\[ E = \begin{cases} 
!(\ast(first1 + n) == \ast(first2 + n)) & \text{for the overloads with no parameter pred;} \\
pred(\ast(first1 + n), \ast(first2 + n)) == \text{false} & \text{for the overloads with a parameter pred and no parameter proj;} \\
!\text{invoke(pred, invoke(proj1, \ast(first1 + n)), invoke(proj2, \ast(first2 + n)))} & \text{for the overloads with both parameters pred and proj1.}
\]
Let \( N \) be \( \min(\text{last1} - \text{first1}, \text{last2} - \text{first2}) \).

Returns: \{ \text{first1} + n, \text{first2} + n \}, where \( n \) is the smallest integer in \([0, N)\) such that \( E \) holds, or \( N \) if no such integer exists.

Complexity: At most \( N \) applications of the corresponding predicate and any projections.

### 25.6.11 Equal

[\text{alg.equal}]

\[
\begin{align*}
\text{template<} &\text{class InputIterator1, class InputIterator2> } \\
&\text{constexpr bool equal(InputIterator1 first1, InputIterator1 last1,} \\
&\text{InputIterator2 first2, InputIterator2 last2);} \\
\text{template<} &\text{class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2> } \\
&\text{bool equal(ExecutionPolicy&& exec,} \\
&\text{ForwardIterator1 first1, ForwardIterator1 last1,} \\
&\text{ForwardIterator2 first2, ForwardIterator2 last2);} \\
\text{template<} &\text{class InputIterator1, class InputIterator2,} \\
&\text{class BinaryPredicate> } \\
&\text{constexpr bool equal(InputIterator1 first1, InputIterator1 last1,} \\
&\text{InputIterator2 first2, InputIterator2 last2, BinaryPredicate pred);} \\
\text{template<} &\text{class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,} \\
&\text{class BinaryPredicate> } \\
&\text{bool equal(ExecutionPolicy&& exec,} \\
&\text{ForwardIterator1 first1, ForwardIterator1 last1,} \\
&\text{ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);} \\
\text{template<} &\text{class InputIterator1, class InputIterator2,} \\
&\text{class BinaryPredicate> } \\
&\text{constexpr bool equal(InputIterator1 first1, InputIterator1 last1,} \\
&\text{InputIterator2 first2, InputIterator2 last2, BinaryPredicate pred);} \\
\text{template<} &\text{class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,} \\
&\text{class BinaryPredicate> } \\
&\text{bool equal(ExecutionPolicy&& exec,} \\
&\text{ForwardIterator1 first1, ForwardIterator1 last1,} \\
&\text{ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);} \\
\text{template<} &\text{input_iterator I1, sentinel_for<}I1\text{> S1, input_iterator I2, sentinel_for<}I2\text{> S2,} \\
&\text{class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity> } \\
&\text{requires indirectly_comparable<}I1\text{, I2, Pred, Proj1, Proj2> } \\
&\text{constexpr bool ranges::equal(I1 first1, S1 last1, I2 first2, S2 last2,} \\
&\text{Pred pred = {}}, \\
&\text{Proj1 proj1 = {}}, \text{Proj2 proj2 = {});} \\
\text{template<} &\text{input_range R1, input_range R2, class Pred = ranges::equal_to,} \\
&\text{class Proj1 = identity, class Proj2 = identity> } \\
&\text{requires indirectly_comparable<}\text{iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2> } \\
&\text{constexpr bool ranges::equal(R1&& r1, R2&& r2, Pred pred = {}),} \\
&\text{Proj1 proj1 = {}}, \text{Proj2 proj2 = {}};) \\
\end{align*}
\]

Let:

(1.1) \( \text{last2 be first2} + (\text{last1} - \text{first1}) \) for the overloads with no parameter \( \text{last2} \) or \( r2 \);

(1.2) \( \text{pred be equal_to{}} \) for the overloads with no parameter \( \text{pred} \);

(1.3) \( \text{E be:} \)

(1.3.1) \( \text{pred(*i, *(first2 + (i - first1)))} \) for the overloads with no parameter \( \text{proj1} \);
invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1)))) for the overloads with parameter proj1.

2 Returns: If last1 - first1 != last2 - first2, return false. Otherwise return true if E holds for every iterator i in the range [first1, last1) Otherwise, returns false.

3 Complexity: If the types of first1, last1, first2, and last2:

- meet the Cpp17RandomAccessIterator requirements (23.3.5.7) for the overloads in namespace std;
- pairwise model sized_sentinel_for (23.3.4.8) for the overloads in namespace ranges,
  and last1 - first1 != last2 - first2, then no applications of the corresponding predicate and each projection; otherwise,

(3.1) — For the overloads with no ExecutionPolicy, at most min(last1 - first1, last2 - first2) applications of the corresponding predicate and any projections.

(3.2) — For the overloads with an ExecutionPolicy, $O(\min(last1 - first1, last2 - first2))$ applications of the corresponding predicate.

25.6.12 Is permutation

template<class ForwardIterator1, class ForwardIterator2>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);

1 Let last2 be first2 + (last1 - first1) for the overloads with no parameter named last2, and let pred be equal_to{} for the overloads with no parameter pred.

2 Mandates: ForwardIterator1 and ForwardIterator2 have the same value type.

3 Preconditions: The comparison function is an equivalence relation.

4 Returns: If last1 - first1 != last2 - first2, return false. Otherwise return true if there exists a permutation of the elements in the range [first2, last2), beginning with ForwardIterator2 begin, such that equal(first1, last1, begin, pred) returns true; otherwise, returns false.

5 Complexity: No applications of the corresponding predicate if ForwardIterator1 and ForwardIterator2 meet the requirements of random access iterators and last1 - first1 != last2 - first2. Otherwise, exactly last1 - first1 applications of the corresponding predicate if equal(first1, last1, first2, last2, pred) would return true; otherwise, at worst $O(N^2)$, where $N$ has the value last1 - first1.
Returns: If last1 - first1 != last2 - first2, return false. Otherwise return true if there exists a permutation of the elements in the range [first2, last2), bounded by [pfirst, plast), such that ranges::equal(first1, last1, pfirst, plast, pred, proj1, proj2) returns true; otherwise, returns false.

Complexity: No applications of the corresponding predicate and projections if:

- S1 and I1 model sized_sentinel_for<S1, I1>,
- S2 and I2 model sized_sentinel_for<S2, I2>, and
- last1 - first1 != last2 - first2.

Otherwise, exactly last1 - first1 applications of the corresponding predicate and projections if ranges::equal(first1, last1, first2, last2, pred, proj1, proj2) would return true; otherwise, at worst $O(N^2)$, where $N$ has the value last1 - first1.

25.6.13 Search

template<class ForwardIterator1, class ForwardIterator2>
constexpr ForwardIterator1
search(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
search(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
constexpr ForwardIterator1
search(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
ForwardIterator1
search(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);

Returns: The first iterator $i$ in the range [first1, last1 - (last2 - first2)) such that for every non-negative integer $n$ less than last2 - first2 the following corresponding conditions hold: $*(i + n) == *(first2 + n)$, pred($*(i + n)$, $*(first2 + n)$) != false. Returns first1 if [first2, last2) is empty, otherwise returns last1 if no such iterator is found.

Complexity: At most (last1 - first1) * (last2 - first2) applications of the corresponding predicate.
— \(i, i + (last2 - first2))\), where \(i\) is the first iterator in the range \([first1, last1 - (last2 - first2))\) such that for every non-negative integer \(n\) less than \(last2 - first2\) the condition

\[
bool\left(\text{invoke}\left(\text{pred}, \text{invoke}\left(\text{proj1}, *(i + n)\right), \text{invoke}\left(\text{proj2}, *(first2 + n)\right)\right)\right) \\text{is true.}
\]

— Returns \([last1, last1]\) if no such iterator exists.

**Complexity:** At most \((last1 - first1) \times (last2 - first2)\) applications of the corresponding predicate and projections.

```cpp
template<class ForwardIterator, class Size, class T>
constexpr ForwardIterator
search_n(ForwardIterator first, ForwardIterator last,
        Size count, const T& value);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
ForwardIterator
search_n(ExecutionPolicy&& exec,
         ForwardIterator first, ForwardIterator last,
         Size count, const T& value);
```

```cpp
template<class ForwardIterator, class Size, class T,
         class BinaryPredicate>
constexpr ForwardIterator
search_n(ForwardIterator first, ForwardIterator last,
         Size count, const T& value,
         BinaryPredicate pred);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Size, class T,
         class BinaryPredicate>
ForwardIterator
search_n(ExecutionPolicy&& exec,
         ForwardIterator first, ForwardIterator last,
         Size count, const T& value,
         BinaryPredicate pred);
```

**Mandates:** The type \(\text{Size}\) is convertible to an integral type (7.3.9, 11.4.8).

**Returns:** The first iterator \(i\) in the range \([first, last - count)\) such that for every non-negative integer \(n\) less than \(count\) the following corresponding conditions hold: \(*(i + n) == value, pred(*\((i + n)), value) \neq false\). Returns \(last\) if no such iterator is found.

**Complexity:** At most \(last - first\) applications of the corresponding predicate.

```cpp
template<forward_iterator I, sentinel_for<I> S, class T,
         class Pred = ranges::equal_to, class Proj = identity>
requires indirectly_comparable<I, const T*, Pred, Proj>
constexpr subrange<I>
ranges::search_n(I first, S last, iter_difference_t<I> count,
                 const T& value, Pred pred = {}, Proj proj = {});
```

```cpp
template<forward_range R, class T, class Pred = ranges::equal_to,
         class Proj = identity>
requires indirectly_comparable<iterator_t<R>, const T*, Pred, Proj>
constexpr borrowed_subrange_t<R>
ranges::search_n(R&& r, range_difference_t<R> count,
                 const T& value, Pred pred = {}, Proj proj = {});
```

**Returns:** \(\{i, i + count\}\) where \(i\) is the first iterator in the range \([first, last - count)\) such that for every non-negative integer \(n\) less than \(count\), the following condition holds: \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *(i + n)), value)\). Returns \(\{last, last\}\) if no such iterator is found.

**Complexity:** At most \(last - first\) applications of the corresponding predicate and projection.

```cpp
template<class ForwardIterator, class Searcher>
constexpr ForwardIterator
search(ForwardIterator first, ForwardIterator last, const Searcher& searcher);
```

**Effects:** Equivalent to: \(\text{return }\text{searcher(first, last).first}\);

**Remarks:** Searcher need not meet the `Cpp17CopyConstructible requirements.
25.7 Mutating sequence operations

25.7.1 Copy

```cpp
template<class InputIterator, class OutputIterator>
constexpr OutputIterator copy(InputIterator first, InputIterator last,
                                OutputIterator result);
```

```cpp
template<input_iterator I, sentinel_for<I> S, weakly_incrementable O>
requires indirectly_copyable<I, O>
constexpr ranges::copy_result<I, O> ranges::copy(I first, S last, O result);
```

```cpp
template<input_range R, weakly_incrementable O>
requires indirectly_copyable<iterator_t<R>, O>
constexpr ranges::copy_result<borrowed_iterator_t<R>, O> ranges::copy(R&& r, O result);
```

1. Let \( N \) be \( \text{last} - \text{first} \).
2. **Preconditions**: \( \text{result} \) is not in the range \([\text{first}, \text{last})\).
3. **Effects**: Copies elements in the range \([\text{first}, \text{last})\) into the range \([\text{result}, \text{result} + N)\) starting from \(\text{first}\) and proceeding to \(\text{last}\). For each non-negative integer \( n < N \), performs \( *(\text{result} + n) = *(\text{first} + n) \).
4. **Returns**: (4.1) \( \text{result} + N \) for the overload in namespace \( \text{std} \). (4.2) \{last, result + N\} for the overloads in namespace \( \text{ranges} \).
5. **Complexity**: Exactly \( N \) assignments.

```cpp
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(ExecutionPolicy&& policy,
                      ForwardIterator1 first, ForwardIterator1 last,
                      ForwardIterator2 result);
```

6. **Preconditions**: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) do not overlap.
7. **Effects**: Copies elements in the range \([\text{first}, \text{last})\) into the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\). For each non-negative integer \( n < (\text{last} - \text{first}) \), performs \( *(\text{result} + n) = *(\text{first} + n) \).
8. **Returns**: \( \text{result} + (\text{last} - \text{first}) \).
9. **Complexity**: Exactly \( \text{last} - \text{first} \) assignments.

```cpp
template<class InputIterator, class Size, class OutputIterator>
constexpr OutputIterator copy_n(InputIterator first, Size n,
                                 OutputIterator result);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator1, class Size, class ForwardIterator2>
ForwardIterator2 copy_n(ExecutionPolicy&& exec,
                        ForwardIterator1 first, Size n,
                        ForwardIterator2 result);
```

10. Let \( N \) be \( \max(0,n) \).
11. **Mandates**: The type \( \text{Size} \) is convertible to an integral type (7.3.9, 11.4.8).
12. **Effects**: For each non-negative integer \( i < N \), performs \( *(\text{result} + i) = *(\text{first} + i) \).
13. **Returns**: (13.1) \( \text{result} + N \) for the overloads in namespace \( \text{std} \). (13.2) \{first + N, result + N\} for the overload in namespace \( \text{ranges} \).
14. **Complexity**: Exactly \( N \) assignments.
15 Let $E$ be:

(15.1) \[ \text{bool(pred(*i))} \] for the overloads in namespace \text{std};

(15.2) \[ \text{bool(invoke(pred, invoke(proj, *i)))} \] for the overloads in namespace \text{ranges},

and $N$ be the number of iterators $i$ in the range $[\text{first}, \text{last})$ for which the condition $E$ holds.

16 \text{Preconditions:} The ranges $[\text{first}, \text{last})$ and $[\text{result}, \text{result} + (\text{last} - \text{first})]$ do not overlap.

[\text{Note 1:} For the overload with an \text{ExecutionPolicy}, there might be a performance cost if \text{iterator_traits<ForwardIterator1>::value_type} is not \text{Cpp17MoveConstructible} (Table 28). — end note]

17 \text{Effects:} Copies all of the elements referred to by the iterator $i$ in the range $[\text{first}, \text{last})$ for which $E$ is true.

18 \text{Returns:}

(18.1) \[ \text{result} + N \] for the overloads in namespace \text{std}.

(18.2) \[ \{last, result + N\} \] for the overloads in namespace \text{ranges}.

19 \text{Complexity:} Exactly $\text{last} - \text{first}$ applications of the corresponding predicate and any projection.

20 \text{Remarks:} Stable (16.4.6.8).

21 \text{template<bidirectional_iterator I1, bidirectional_iterator I2>}
\text{constexpr BidirectionalIterator2 copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last, bidirectional_iterator I2 result);}
template<class InputIterator, class OutputIterator>
constexpr OutputIterator move(InputIterator first, InputIterator last,
    OutputIterator result);

template<input_iterator I, sentinel_for<I> S, weakly_incrementable O>
requires indirectly_movable<I, O>
constexpr ranges::move_result<I, O>
ranges::move(I first, S last, O result);

template<input_range R, weakly_incrementable O>
requires indirectly_movable<iterator_t<R>, O>
constexpr ranges::move_result<borrowed_iterator_t<R>, O>
ranges::move(R&& r, O result);

Let \( E \) be

1. \( \text{std::move(* (first + n))} \) for the overload in namespace \( \text{std} \);
2. \( \text{ranges::iter_move(first + n)} \) for the overloads in namespace \( \text{ranges} \).

Let \( N \) be \( \text{last - first} \).

Preconditions: result is not in the range \([\text{first}, \text{last})\).

Effects: Moves elements in the range \([\text{first}, \text{last})\) into the range \([\text{result}, \text{result} + N)\) starting from first and proceeding to last. For each non-negative integer \( n < N \), performs \( * (\text{result} + n) = E \).

Returns: result + \( N \).

Complexity: Exactly \( N \) assignments.

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 move(ExecutionPolicy&& policy,
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 result);

Let \( N \) be \( \text{last - first} \).

Preconditions: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + N)\) do not overlap.

Effects: Moves elements in the range \([\text{first}, \text{last})\) into the range \([\text{result}, \text{result} + N)\). For each non-negative integer \( n < N \), performs \( * (\text{result} + n) = \text{std::move(* (first + n))} \).

Returns: result + \( N \).

Complexity: Exactly \( N \) assignments.

template<class BidirectionalIterator1, class BidirectionalIterator2>
constexpr BidirectionalIterator2
move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
    BidirectionalIterator2 result);

template<bidirectional_iterator I1, sentinel_for<I1> S1, bidirectional_iterator I2>
requires indirectly_movable<I1, I2>
constexpr ranges::move_backward_result<I1, I2>
ranges::move_backward(I1 first, S1 last, I2 result);

template<bidirectional_range R, bidirectional_iterator I>
requires indirectly_movable<iterator_t<R>, I>
constexpr ranges::move_backward_result<borrowed_iterator_t<R>, I>
ranges::move_backward(R&& r, I result);

Let \( E \) be

— \( \text{result} - N \) for the overload in namespace \( \text{std} \).
— \( \{ \text{last}, \text{result} - N \} \) for the overloads in namespace \( \text{ranges} \).

Complexity: Exactly \( N \) assignments.

§ 25.7.2 Move
Let \( N \) be \( \text{last} - \text{first} \).

**Preconditions:** result is not in the range \([\text{first}, \text{last}]\).

**Effects:** Moves elements in the range \([\text{first}, \text{last})\) into the range \([\text{result} - N, \text{result})\) starting from \( \text{last} - 1 \) and proceeding to \( \text{first} \).

**Returns:**

- \( \text{result} - N \) for the overload in namespace \text{std}.
- \( \{\text{last}, \text{result} - N\} \) for the overloads in namespace \text{ranges}.

**Complexity:** Exactly \( N \) assignments.

### 25.7.3 Swap

**template<class ForwardIterator1, class ForwardIterator2>**

```cpp
constexpr ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);
```

**template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>**

```cpp
ForwardIterator2 swap_ranges(ExecutionPolicy&& exec, ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);
```

**template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2> requires indirectly_swappable<I1, I2>**

```cpp
ranges::swap_ranges_result<I1, I2> ranges::swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);
```

**template<input_range R1, input_range R2> requires indirectly_swappable<iterator_t<R1>, iterator_t<R2>>**

```cpp
ranges::swap_ranges_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>> ranges::swap_ranges(R1&& r1, R2&& r2);
```

Let:

- \( \text{last2} \) be \( \text{first2} + (\text{last1} - \text{first1}) \) for the overloads with no parameter named \( \text{last2} \);
- \( M \) be \( \min(\text{last1} - \text{first1}, \text{last2} - \text{first2}) \).

**Preconditions:** The two ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) do not overlap. For the overloads in namespace \text{std}, \(*(\text{first1} + n)* \) is swappable with \((16.4.4.3)\) \(*(\text{first2} + n)* \).

**Effects:** For each non-negative integer \( n < M \) performs:

- \( \text{swap}(*(\text{first1} + n), *(\text{first2} + n)) \) for the overloads in namespace \text{std};
- \( \text{ranges::iter_swap}(\text{first1} + n, \text{first2} + n) \) for the overloads in namespace \text{ranges}.

**Returns:**

- \( \text{last2} \) for the overloads in namespace \text{std}.
- \( \{\text{first1} + M, \text{first2} + M\} \) for the overloads in namespace \text{ranges}.

**Complexity:** Exactly \( M \) swaps.

**template<class ForwardIterator1, class ForwardIterator2>**

```cpp
constexpr void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

**Preconditions:** \( a \) and \( b \) are dereferenceable. \(*a\) is swappable with \((16.4.4.3)\) \(*b*\).

**Effects:** As if by \( \text{swap}(*a, *b) \).

---

\(^{238}\) move\_backward can be used instead of move when \( \text{last} \) is in the range \([\text{result} - N, \text{result})\).
25.7.4 Transform

```cpp
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
constexpr OutputIterator
transform(InputIterator first1, InputIterator last1,
          OutputIterator result, UnaryOperation op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
ForwardIterator2
transform(ExecutionPolicy&& exec,
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 result, UnaryOperation op);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class BinaryOperation>
constexpr OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2, OutputIterator result,
         BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class BinaryOperation>
ForwardIterator
transform(ExecutionPolicy&& exec,
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator result,
          BinaryOperation binary_op);

template<input_iterator I, sentinel_for<I> S, weakly_incrementable O,
          copy_constructible F, class Proj = identity>
requires indirectly_writable<O, indirect_result_t<F&, projected<I, Proj>>
constexpr ranges::unary_transform_result<I, O>
ranges::transform(I first1, S last1, O result, F op, Proj proj = {});

template<input_range R, weakly_incrementable O, copy_constructible F,
          class Proj = identity>
requires indirectly_writable<O, indirect_result_t<F&, projected<iterator_t<R>, Proj>>
constexpr ranges::unary_transform_result<borrowed_iterator_t<R>, O>
ranges::transform(R&& r, O result, F op, Proj proj = {});

template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
          weakly_incrementable O, copy_constructible F, class Proj1 = identity,
          class Proj2 = identity>
requires indirectly_writable<O, indirect_result_t<F&, projected<I1, Proj1>,
                                    projected<I2, Proj2>>, copy_constructible F, class Proj1 = identity,
          class Proj2 = identity>
requires indirectly_writable<O, indirect_result_t<F&, projected<iterator_t<I1>, Proj1>,
                                    projected<iterator_t<I2>, Proj2>>, copy_constructible F, class Proj1 = identity,
          class Proj2 = identity>
constexpr ranges::binary_transform_result<borrowed_iterator_t<I1>, borrowed_iterator_t<I2>, O>
ranges::transform(R1&& r1, R2&& r2, O result,
                   F binary_op, Proj1 proj1 = {}, Proj2 proj2 = {});
```

Let:

1. last2 be first2 + (last1 - first1) for the overloads with parameter first2 but no parameter last2;
2. \( N \) be last1 - first1 for unary transforms, or \( \min(last1 - first1, last2 - first2) \) for binary transforms;
3. \( E \) be \( \text{op}(*(first1 + (i - result))) \) for unary transforms defined in namespace std;
— binary_op(*(first1 + (i - result)), *(first2 + (i - result))) for binary transforms defined in namespace std;

— invoke(op, invoke(proj), *(first1 + (i - result))) for unary transforms defined in namespace std;

— invoke(binary_op, invoke(proj1, *(first1 + (i - result))), invoke(proj2, *(first2 + (i - result)))) for binary transforms defined in namespace ranges;

**Preconditions:** op and binary_op do not invalidate iterators or subranges, nor modify elements in the ranges

— [first1, first1 + N],
— [first2, first2 + N], and
— [result, result + N].

**Effects:** Assigns through every iterator i in the range [result, result + N) a new corresponding value equal to E.

**Returns:**

— result + N for the overloads defined in namespace std.
— {first1 + N, result + N} for unary transforms defined in namespace ranges.
— {first1 + N, first2 + N, result + N} for binary transforms defined in namespace ranges.

**Complexity:** Exactly N applications of op or binary_op, and any projections. This requirement also applies to the overload with an ExecutionPolicy.

**Remarks:** result may be equal to first1 or first2.

### 25.7.5 Replace

```
25.7.5 Replace [alg.replace]

template<class ForwardIterator, class T>
constexpr void replace(ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value);

template<class ExecutionPolicy, class ForwardIterator, class T>
void replace(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
constexpr void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value);

template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
void replace_if(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value);

template<input_iterator I, sentinel_for<I> S, class T1, class T2, class Proj = identity>
requires indirectly_writable<I, const T2&> && indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T1*>&
constexpr I ranges::replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = {});

template<input_range R, class T1, class T2, class Proj = identity>
requires indirectly_writable<iterator_t<R>, const T2&> && indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T1*>&
constexpr borrowed_iterator_t<R> ranges::replace(R&& r, const T1& old_value, const T2& new_value, Proj proj = {});

template<input_iterator I, sentinel_for<I> S, class T, class Proj = identity, indirect_unary_predicate<projected<I, Proj>> Pred>
requires indirectly_writable<I, const T&>
constexpr I ranges::replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = {});
```

239) The use of fully closed ranges is intentional.
Let \( E \) be

1. \( \text{bool}(\ast i == \text{old\_value}) \) for \( \text{replace} \);
2. \( \text{bool}(\text{pred}(\ast i)) \) for \( \text{replace\_if} \);
3. \( \text{bool}(	ext{invoke}(\text{proj}, \ast i) == \text{old\_value}) \) for \( \text{ranges}\_\text{replace} \);
4. \( \text{bool}(	ext{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i))) \) for \( \text{ranges}\_\text{replace\_if} \).

**Mandates:** \( \text{new\_value} \) is writable (23.3.1) to first.

**Effects:** Substitutes elements referred by the iterator \( i \) in the range \([\text{first}, \text{last})\) with \( \text{new\_value} \), when \( E \) is true.

**Returns:** last for the overloads in namespace \( \text{ranges} \).

**Complexity:** Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate and any projection.
Let \( E \) be

\[
\begin{align*}
(6.1) & \quad \text{bool}(*\text{first} + (i - \text{result})) == \text{old_value} \quad \text{for replace_copy} \\
(6.2) & \quad \text{bool}(\text{pred}(*\text{first} + (i - \text{result}))) \quad \text{for replace_copy_if} \\
(6.3) & \quad \text{bool}(\text{invoke}(<\text{proj}, *\text{first} + (i - \text{result}))} == \text{old_value} \quad \text{for ranges::replace_copy} \\
(6.4) & \quad \text{bool}(\text{invoke}(<\text{pred}, \text{invoke}(<\text{proj}, *\text{first} + (i - \text{result})))) \quad \text{for ranges::replace_copy_if}.
\end{align*}
\]

**Mandates:** The results of the expressions *first and new_value are writable (23.3.1) to result.

**Preconditions:** The ranges [first, last) and [result, result + (last - first)) do not overlap.

**Effects:** Assigns through every iterator \( i \) in the range [result, result + (last - first)) a new corresponding value

\[
\begin{align*}
(9.1) & \quad \text{new_value if } E \text{ is true or} \\
(9.2) & \quad *(\text{first} + (i - \text{result})) \text{ otherwise.}
\end{align*}
\]

**Returns:**

\[
\begin{align*}
(10.1) & \quad \text{result + (last - first)} \text{ for the overloads in namespace std.} \\
(10.2) & \quad \{\text{last, result + (last - first)}\} \text{ for the overloads in namespace ranges.}
\end{align*}
\]

**Complexity:** Exactly last - first applications of the corresponding predicate and any projection.

### 25.7.6 Fill

[alg.fill]

```cpp
ranges::replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
    Proj proj = {});
```

```cpp
template<input_range R, class T, output_iterator<const T&> O, class Proj = identity,
    indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
    requires indirectly_copyable<iterator_t<R>, O>
constexpr ranges::replace_copy_if_result<borrowed_iterator_t<R>, O>
ranges::replace_copy_if(R&& r, O result, Pred pred, const T& new_value,
    Proj proj = {});
```

```cpp
Let \( E \) be

\[
\begin{align*}
(6.1) & \quad \text{bool}(*\text{first} + (i - \text{result})) == \text{old_value} \quad \text{for replace_copy} \\
(6.2) & \quad \text{bool}(\text{pred}(*\text{first} + (i - \text{result}))) \quad \text{for replace_copy_if} \\
(6.3) & \quad \text{bool}(\text{invoke}(<\text{proj}, *\text{first} + (i - \text{result}))} == \text{old_value} \quad \text{for ranges::replace_copy} \\
(6.4) & \quad \text{bool}(\text{invoke}(<\text{pred}, \text{invoke}(<\text{proj}, *\text{first} + (i - \text{result})))) \quad \text{for ranges::replace_copy_if}.
\end{align*}
\]

**Mandates:** The results of the expressions *first and new_value are writable (23.3.1) to result.

**Preconditions:** The ranges [first, last) and [result, result + (last - first)) do not overlap.

**Effects:** Assigns through every iterator \( i \) in the range [result, result + (last - first)) a new corresponding value

\[
\begin{align*}
(9.1) & \quad \text{new_value if } E \text{ is true or} \\
(9.2) & \quad *(\text{first} + (i - \text{result})) \text{ otherwise.}
\end{align*}
\]

**Returns:**

\[
\begin{align*}
(10.1) & \quad \text{result + (last - first)} \text{ for the overloads in namespace std.} \\
(10.2) & \quad \{\text{last, result + (last - first)}\} \text{ for the overloads in namespace ranges.}
\end{align*}
\]

**Complexity:** Exactly last - first applications of the corresponding predicate and any projection.

### 25.7.6 Fill

[alg.fill]
25.7.7 Generate

```
[alg.generate]
template<class ForwardIterator, class Generator>
constexpr void generate(ForwardIterator first, ForwardIterator last,
        Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Generator>
void generate(ExecutionPolicy&& exec,
        ForwardIterator first, ForwardIterator last,
        Generator gen);

template<class OutputIterator, class Size, class Generator>
constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
ForwardIterator generate_n(ExecutionPolicy&& exec,
        ForwardIterator first, Size n, Generator gen);

template<input_or_output_iterator O, sentinel_for<O> S, copy_constructible F>
requires invocable<F&> && indirectly_writable<O, invoke_result_t<F&>>
constexpr O ranges::generate(O first, S last, F gen);
template<class R, copy_constructible F>
requires invocable<F&> && output_range<R, invoke_result_t<F&>>
constexpr borrowed_iterator_t<R> ranges::generate(R&& r, F gen);
```  

Let $N$ be $\max(0, n)$ for the generate_n algorithms, and $\text{last} - \text{first}$ for the generate algorithms.

1. **Mandates:** Size is convertible to an integral type (7.3.9, 11.4.8).
2. **Effects:** Assigns the result of successive evaluations of $\text{gen}()$ through each iterator in the range $[\text{first}, \text{first} + N)$.
3. **Returns:** $\text{first} + N$.
4. **Complexity:** Exactly $N$ evaluations of $\text{gen}()$ and assignments.

25.7.8 Remove

```
[alg.remove]
template<class ForwardIterator, class T>
constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
        const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
ForwardIterator remove(ExecutionPolicy&& exec,
        ForwardIterator first, ForwardIterator last,
        const T& value);

template<class ForwardIterator, class Predicate>
constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
        Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator remove_if(ExecutionPolicy&& exec,
        ForwardIterator first, ForwardIterator last,
        Predicate pred);

template<permutable I, sentinel_for<I> S, class T, class Proj = identity>
requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>>, const T*>
constexpr subrange<I> ranges::remove(I first, S last, const T& value, Proj proj = {});
template<form_range R, class T, class Proj = identity>
requires permutable<iterator_t<R>> &&
    indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>>, Proj>, const T*>
constexpr borrowed_subrange_t<R> ranges::remove(R&& r, const T& value, Proj proj = {});
```
template<forward_range R, class Proj = identity, 
    indirect Unary predicate<projected<iterator_t<R>, Proj>> Pred>
    requires permutable<iterator_t<R>>
    constexpr borrowed subrange_t<R>
    ranges::remove_if(R&& r, Pred pred, Proj proj = {});

Let $E$ be

1. $\text{bool}(\ast i == \text{value})$ for remove;
2. $\text{bool}(\text{pred}(\ast i))$ for remove_if;
3. $\text{bool}(\text{invoke}(\text{proj}, \ast i) == \text{value})$ for ranges::remove;
4. $\text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i)))$ for ranges::remove_if.

Preconditions: For the algorithms in namespace \texttt{std}, the type of *first meets the Cpp17MoveAssignble requirements (Table 30).

Effects: Eliminates all the elements referred to by iterator i in the range \([\text{first}, \text{last})\) for which $E$ holds.

Returns: Let $j$ be the end of the resulting range. Returns:

1. $j$ for the overloads in namespace \texttt{std}.
2. $\{j, \text{last}\}$ for the overloads in namespace \texttt{ranges}.

Complexity: Exactly \texttt{last} - \texttt{first} applications of the corresponding predicate and any projection.

Remarks: Stable (16.4.6.8).

[Note 1: Each element in the range \([\text{ret}, \text{last})\), where \texttt{ret} is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range. — end note]
template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, 
class Proj = identity, indirect_unary_predicate<projected<I, Proj>> Pred>
  requires indirectly_copyable<I, O>
  constexpr ranges::remove_copy_if_result<I, O>
  ranges::remove_copy_if(I first, S last, O result, Pred pred, Proj proj = {});  

Let $E$ be

- $\text{bool}(*i == \text{value})$ for remove_copy;
- $\text{bool}(\text{pred}(\ast i))$ for remove_copy_if;
- $\text{bool}((\text{invoke}(\text{proj}, \ast i) == \text{value})$ for ranges::remove_copy;
- $\text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i)))$ for ranges::remove_copy_if.

Let $N$ be the number of elements in $[\text{first}, \text{last})$ for which $E$ is false.

**Mandates:** $\ast $first is writable (23.3.1) to result.

** Preconditions:** The ranges $[\text{first}, \text{last})$ and $[\text{result}, \text{result} + (\text{last} - \text{first}))$ do not overlap.

[Note 2: For the overloads with an ExecutionPolicy, there might be a performance cost if iterator_traits<ForwardIterator>::value_type does not meet the Cpp17MoveConstructible (Table 28) requirements. —end note]

**Effects:** Copies all the elements referred to by the iterator $i$ in the range $[\text{first}, \text{last})$ for which $E$ is false.

**Returns:**

- $\text{result} + N$, for the algorithms in namespace std.
- $\{\text{last}, \text{result} + N\}$, for the algorithms in namespace ranges.

**Complexity:** Exactly last - first applications of the corresponding predicate and any projection.

**Remarks:** Stable (16.4.6.8).

25.7.9 Unique

```
§ 25.7.9 1109
```
(1.2) — bool(invoke(comp, invoke(proj, *(i - 1)), invoke(proj, *i))) for the overloads in namespace ranges.

Preconditions: For the overloads in namespace std, pred is an equivalence relation and the type of *first meets the Cpp17MoveAssignable requirements (Table 30).

Effects: For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator i in the range [first + 1, last) for which E is true.

Returns: Let j be the end of the resulting range. Returns:
— j for the overloads in namespace std.
— {j, last} for the overloads in namespace ranges.

Complexity: For nonempty ranges, exactly (last - first) - 1 applications of the corresponding predicate and no more than twice as many applications of any projection.

```cpp
template<class InputIterator, class OutputIterator>
constexpr OutputIterator
unique_copy(InputIterator first, InputIterator last,
OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
unique_copy(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result);
```

6 Let pred be equal_to{} for the overloads in namespace std with no parameter pred, and let E be

(6.1) — bool(pred(*i, *(i - 1))) for the overloads in namespace std;
(6.2) — bool(invoke(comp, invoke(proj, *i), invoke(proj, *(i - 1)))) for the overloads in namespace ranges.

Mandates: *first is writable (23.3.1) to result.

Preconditions:

(8.1) — The ranges [first, last) and [result, result+(last-first)) do not overlap.
For the overloads in namespace std:

— The comparison function is an equivalence relation.

For the overloads with no ExecutionPolicy, let T be the value type of InputIterator. If InputIterator meets the Cpp17ForwardIterator requirements, then there are no additional requirements for T. Otherwise, if OutputIterator meets the Cpp17ForwardIterator requirements and its value type is the same as T, then T meets the Cpp17CopyAssignable (Table 31) requirements. Otherwise, T meets both the Cpp17CopyConstructible (Table 29) and Cpp17CopyAssignable requirements.

[Note 1: For the overloads with an ExecutionPolicy, there might be a performance cost if the value type of ForwardIterator1 does not meet both the Cpp17CopyConstructible and Cpp17CopyAssignable requirements. — end note]

Effects: Copies only the first element from every consecutive group of equal elements referred to by the iterator i in the range [first, last) for which E holds.

Returns:

— result + N for the overloads in namespace std.

— {last, result + N} for the overloads in namespace ranges.

Complexity: Exactly last - first - 1 applications of the corresponding predicate and no more than twice as many applications of any projection.
template<bidirectional_range R, weakly_incrementable O>
requires indirectly_copyable<iterator_t<R>, O>
constexpr ranges::reverse_copy_result<borrowed_iterator_t<R>, O>
ranges::reverse_copy(R&& r, O result);

Let \( N \) be \( \text{last} - \text{first} \).

**Preconditions:** The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + N)\) do not overlap.

**Effects:** Copies the range \([\text{first}, \text{last})\) to the range \([\text{result}, \text{result} + N)\) such that for every non-negative integer \( i < N \) the following assignment takes place: \( \text{*}((\text{result} + N - 1 - i) = \text{*}((\text{first} + i)) \).

**Returns:**

- \( \text{result} + N \) for the overloads in namespace \text{std}.
- \( \{\text{last}, \text{result} + N)\) for the overloads in namespace \text{ranges}.

**Complexity:** Exactly \( N \) assignments.

---

25.7.11 Rotate

| template<class ForwardIterator>
| constexpr ForwardIterator
| rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);

| template<class ExecutionPolicy, class ForwardIterator>
| ForwardIterator
| rotate(ExecutionPolicy&& exec,
| ForwardIterator first, ForwardIterator middle, ForwardIterator last);

| template<permutable I, sentinel_for<I> S>
| constexpr subrange<I> ranges::rotate(I first, I middle, S last);

**Preconditions:** \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\) are valid ranges. For the overloads in namespace \text{std}. \text{ForwardIterator} meets the \text{Cpp17ValueSwappable} requirements (16.4.4.3), and the type of \text{*}first meets the \text{Cpp17MoveConstructible} (Table 28) and \text{Cpp17MoveAssignble} (Table 30) requirements.

**Effects:** For each non-negative integer \( i < (\text{last} - \text{first}) \), places the element from the position \( \text{first} + i \) into position \( \text{first} + (i + (\text{last} - \text{middle})) \% (\text{last} - \text{first}) \).

[Note 1: This is a left rotate. — end note]

**Returns:**

- \( \text{first} + (\text{last} - \text{middle}) \) for the overloads in namespace \text{std}.
- \( \{\text{first} + (\text{last} - \text{middle}), \text{last})\) for the overload in namespace \text{ranges}.

**Complexity:** At most \( \text{last} - \text{first} \) swaps.

| template<forward_range R>
| constexpr borrowed subrange_t<R> ranges::rotate(R&& r, iterator_t<R> middle);

**Effects:** Equivalent to: return ranges::rotate(ranges::begin(r), middle, ranges::end(r));

| template<class ForwardIterator, class OutputIterator>
| constexpr OutputIterator
| rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last,
| OutputIterator result);

| template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
| ForwardIterator2
| rotate_copy(ExecutionPolicy&& exec,
| ForwardIterator1 first, ForwardIterator1 middle, ForwardIterator1 last,
| ForwardIterator2 result);

| template<forward_iterator I, sentinel_for<I> S, weakly_incrementable O>
| constexpr ranges::rotate_copy_result<I, O>
| constexpr ranges::rotate_copy_result<I, O>
ranges::rotate_copy(I first, I middle, S last, O result);

Let \( N \) be \( \text{last} - \text{first} \).

Preconditions: \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\) are valid ranges. The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + N)\) do not overlap.

Effects: Copies the range \([\text{first}, \text{last})\) to the range \([\text{result}, \text{result} + N)\) such that for each non-negative integer \( i < N \) the following assignment takes place:

\[
*(\text{result} + i) = *(\text{first} + (i + (\text{middle} - \text{first})) \mod N).
\]

Returns:

\( (9.1) \)

— \( \text{result} + N \) for the overloads in namespace \( \text{std} \).

\( (9.2) \)

— \( \{\text{last}, \text{result} + N\} \) for the overload in namespace \( \text{ranges} \).

Complexity: Exactly \( N \) assignments.

```cpp
template<forward_range R, weakly_incrementable O>
requires indirectly_copyable<iterator_t<R>, O>
constexpr ranges::rotate_copy_result<borrowed_iterator_t<R>, O>
ranges::rotate_copy(R&& r, iterator_t<R> middle, O result);
```

Effects: Equivalent to:

\[
\text{return ranges::rotate_copy(ranges::begin(r), middle, ranges::end(r), result)};\]

25.7.12 Sample [alg.random.sample]

```cpp
template<class PopulationIterator, class SampleIterator, class Distance, class UniformRandomBitGenerator>
SampleIterator sample(PopulationIterator first, PopulationIterator last, SampleIterator out, Distance n, UniformRandomBitGenerator&& g);
```

```cpp
template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, class Gen>
requires (forward_iterator<I> || random_access_iterator<O>) &&
indirectly_copyable<I, O> &&
uniform_random_bit_generator<remove_reference_t<Gen>>
O ranges::sample(I first, S last, O out, iter_difference_t<I> n, Gen&& g);
```

```cpp
template<input_range R, weakly_incrementable O, class Gen>
requires (forward_range<R> || random_access_iterator<O>) &&
indirectly_copyable<iterator_t<R>, O> &&
uniform_random_bit_generator<remove_reference_t<Gen>>
O ranges::sample(R&& r, O out, range_difference_t<R> n, Gen&& g);
```

Mandates: For the overload in namespace \( \text{std} \), \( \text{Distance} \) is an integer type and \( *\text{first} \) is writable (23.3.1) to out.

Preconditions: \( \text{out} \) is not in the range \([\text{first}, \text{last})\). For the overload in namespace \( \text{std} \):

\( (2.1) \)

— \( \text{PopulationIterator} \) meets the \( \text{Cpp17InputIterator} \) requirements (23.3.5.3).

\( (2.2) \)

— \( \text{SampleIterator} \) meets the \( \text{Cpp17OutputIterator} \) requirements (23.3.5.4).

\( (2.3) \)

— \( \text{SampleIterator} \) meets the \( \text{Cpp17RandomAccessIterator} \) requirements (23.3.5.7) unless \( \text{PopulationIterator} \) meets the \( \text{Cpp17ForwardIterator} \) requirements (23.3.5.5).

\( (2.4) \)

— \( \text{remove_reference_t<UniformRandomBitGenerator>} \) meets the requirements of a uniform random bit generator type (26.6.3.3).

Effects: Copies \( \min(\text{last} - \text{first}, n) \) elements (the \( \text{sample} \)) from \([\text{first}, \text{last})\) (the \( \text{population} \)) to \( \text{out} \) such that each possible sample has equal probability of appearance.

[Note 1: Algorithms that obtain such effects include selection sampling and reservoir sampling. — end note]

Returns: The end of the resulting sample range.

Complexity: \( \Theta(\text{last} - \text{first}) \).

Remarks:
For the overload in namespace `std`, stable if and only if `PopulationIterator` meets the `Cpp17ForwardIterator` requirements. For the first overload in namespace `ranges`, stable if and only if I models `forward_iterator`.

To the extent that the implementation of this function makes use of random numbers, the object g serves as the implementation’s source of randomness.

25.7.13 Shuffle

```cpp
template<class RandomAccessIterator, class UniformRandomBitGenerator>
void shuffle(RandomAccessIterator first, RandomAccessIterator last, UniformRandomBitGenerator&& g);
```

```cpp
template<random_access_iterator I, sentinel_for<I> S, class Gen>
requires permutable<I> && uniform_random_bit_generator<remove_reference_t<Gen>>
I ranges::shuffle(I first, S last, Gen&& g);
```

```cpp
template<random_access_range R, class Gen>
requires permutable<iterator_t<R>> && uniform_random_bit_generator<remove_reference_t<Gen>>
borrowed_iterator_t<R> ranges::shuffle(R&& r, Gen&& g);
```

**Preconditions:** For the overload in namespace std:

1. RandomAccessIterator meets the `Cpp17ValueSwappable` requirements (16.4.4.3).
2. The type `remove_reference_t<UniformRandomBitGenerator>` meets the uniform random bit generator (26.6.3.3) requirements.

**Effects:** Permutes the elements in the range `[first, last)` such that each possible permutation of those elements has equal probability of appearance.

**Returns:** `last` for the overloads in namespace `ranges`.

**Complexity:** At most `(last - first) - 1` swaps.

**Remarks:** To the extent that the implementation of this function makes use of random numbers, the object referenced by g shall serve as the implementation’s source of randomness.

25.7.14 Shift

```cpp
template<class ForwardIterator>
constexpr ForwardIterator shift_left(ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator shift_left(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);
```

**Preconditions:** n >= 0 is true. The type of *first meets the `Cpp17MoveAssignable` requirements.

**Effects:** If n == 0 or n >= last - first, does nothing. Otherwise, moves the element from position first + n + i into position first + i for each non-negative integer i < (last - first) - n. In the first overload case, does so in order starting from i = 0 and proceeding to i = (last - first) - n - 1.

**Returns:** first + (last - first - n) if n < last - first, otherwise first.

**Complexity:** At most (last - first) - n assignments.

```cpp
template<class ForwardIterator>
constexpr ForwardIterator shift_right(ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);
```
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator
    shift_right(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
        typename iterator_traits<ForwardIterator>::difference_type n);

Preconditions: \( n \geq 0 \) is true. The type of \(*first\) meets the Cpp17MoveAssignble requirements. ForwardIterator meets the Cpp17BidirectionalIterator requirements (23.3.5.6) or the Cpp17ValueSwappable requirements.

Effects: If \( n = 0 \) or \( n \geq \text{last} - \text{first} \), does nothing. Otherwise, moves the element from position \( \text{first} + i \) into position \( \text{first} + n + i \) for each non-negative integer \( i < (\text{last} - \text{first}) - n \). In the first overload case, if ForwardIterator meets the Cpp17BidirectionalIterator requirements, does so in order starting from \( i = (\text{last} - \text{first}) - n - 1 \) and proceeding to \( i = 0 \).

Returns: \( \text{first} + n \) if \( n < \text{last} - \text{first} \), otherwise last.

Complexity: At most \( (\text{last} - \text{first}) - n \) assignments or swaps.

25.8 Sorting and related operations

25.8.1 General

The operations in 25.8 defined directly in namespace std have two versions: one that takes a function object of type Compare and one that uses an operator<.

Compare is a function object type (20.14) that meets the requirements for a template parameter named BinaryPredicate (25.2). The return value of the function call operation applied to an object of type Compare, when contextually converted to bool (7.3), yields true if the first argument of the call is less than the second, and false otherwise. Compare comp is used throughout for algorithms assuming an ordering relation.

For all algorithms that take Compare, there is a version that uses operator< instead. That is, comp(*i, *j) != false defaults to *i < *j != false. For algorithms other than those described in 25.8.4, comp shall induce a strict weak ordering on the values.

The term strict refers to the requirement of an irreflexive relation (!comp(x, x) for all x), and the term weak to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define equiv(a, b) as !comp(a, b) && !comp(b, a), then the requirements are that comp and equiv both be transitive relations:

- \( \text{comp}(a, b) \&\& \text{comp}(b, c) \implies \text{comp}(a, c) \) (4.1)
- \( \text{equiv}(a, b) \&\& \text{equiv}(b, c) \implies \text{equiv}(a, c) \) (4.2)

[Note 1: Under these conditions, it can be shown that]

- equiv is an equivalence relation,
- equiv induces a well-defined relation on the equivalence classes determined by equiv, and
- the induced relation is a strict total ordering.

[end note]

A sequence is sorted with respect to a comp and proj for a comparator and projection comp and proj if for every iterator i pointing to the sequence and every non-negative integer n such that \( i + n \) is a valid iterator pointing to an element of the sequence,

\[ \text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, *(i + n)), \text{invoke}(\text{proj}, *i))) \]

is false.

A sequence [start, finish) is partitioned with respect to an expression f(e) if there exists an integer n such that for all \( 0 \leq i < (\text{finish} - \text{start}) \), f(*((\text{start} + i))) is true if and only if \( i < n \).

In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an operator==, but an equivalence relation induced by the strict weak ordering. That is, two elements a and b are considered equivalent if and only if !(a < b) && !(b < a).

25.8.2 Sorting

25.8.2.1 sort

template<class RandomAccessIterator>
constexpr void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
void sort(ExecutionPolicy&& exec,
RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
constexpr void sort(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void sort(ExecutionPolicy&& exec,
RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr I
ranges::sort(I first, S last, Comp comp = {}, Proj proj = {});

template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr borrowed_iterator_t<R>
ranges::sort(R&& r, Comp comp = {}, Proj proj = {});

Let \( \text{comp} \) be \( \text{less}\{} \) and \( \text{proj} \) be \( \text{identity}\{} \) for the overloads with no parameters by those names.

1 Preconditions: For the overloads in namespace std, RandomAccessIterator meets the Cpp17Value-Swappable requirements (16.4.4.3) and the type of \(*\text{first}\) meets the Cpp17MoveConstructible (Table 28) and Cpp17MoveAssignable (Table 30) requirements.

2 Effects: Sorts the elements in the range \([\text{first}, \text{last})\) with respect to \( \text{comp} \) and \( \text{proj} \).

3 Returns: last for the overloads in namespace ranges.

4 Complexity: Let \( N \) be \( \text{last} - \text{first} \). \( O(N \log N) \) comparisons and projections.

25.8.2.2 stable_sort

template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator>
void stable_sort(ExecutionPolicy&& exec,
RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void stable_sort(ExecutionPolicy&& exec,
RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
class Proj = identity>
requires sortable<I, Comp, Proj>
I ranges::stable_sort(I first, S last, Comp comp = {}, Proj proj = {});

template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
borrowed_iterator_t<R>
ranges::stable_sort(R&& r, Comp comp = {}, Proj proj = {});

Let \( \text{comp} \) be \( \text{less}\{} \) and \( \text{proj} \) be \( \text{identity}\{} \) for the overloads with no parameters by those names.

1 Preconditions: For the overloads in namespace std, RandomAccessIterator meets the Cpp17Value-Swappable requirements (16.4.4.3) and the type of \(*\text{first}\) meets the Cpp17MoveConstructible (Table 28) and Cpp17MoveAssignable (Table 30) requirements.

2 Effects: Sorts the elements in the range \([\text{first}, \text{last})\) with respect to \( \text{comp} \) and \( \text{proj} \).

3 Returns: last for the overloads in namespace ranges.
Complexity: Let \( N \) be \( \text{last - first} \). If enough extra memory is available, \( N \log(N) \) comparisons. Otherwise, at most \( N \log^2(N) \) comparisons. In either case, twice as many projections as the number of comparisons.

Remarks: Stable (16.4.6.8).

25.8.2.3 partial_sort

```cpp
template<class RandomAccessIterator>
constexpr void partial_sort(RandomAccessIterator first,
                            RandomAccessIterator middle,
                            RandomAccessIterator last);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator>
void partial_sort(ExecutionPolicy& exec,
                 RandomAccessIterator first,
                 RandomAccessIterator middle,
                 RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
constexpr void partial_sort(RandomAccessIterator first,
                            RandomAccessIterator middle,
                            RandomAccessIterator last,
                            Compare comp);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void partial_sort(ExecutionPolicy&& exec,
                 RandomAccessIterator first,
                 RandomAccessIterator middle,
                 RandomAccessIterator last,
                 Compare comp);
```

```cpp
template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
         class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr I
ranges::partial_sort(I first, I middle, S last, Comp comp = {}, Proj proj = {});
```

Let \( \text{comp} \) be \( \text{less}() \) and \( \text{proj} \) be \( \text{identity}() \) for the overloads with no parameters by those names.

Preconditions: \( [\text{first}, \text{middle}) \) and \( [\text{middle}, \text{last}) \) are valid ranges. For the overloads in namespace \( \text{std} \), \( \text{RandomAccessIterator} \) meets the \( \text{Cpp17ValueSwappable} \) requirements (16.4.4.3) and the type of \( *\text{first} \) meets the \( \text{Cpp17MoveConstructible} \) (Table 28) and \( \text{Cpp17MoveAssignable} \) (Table 30) requirements.

Effects: Places the first \( \text{middle - first} \) elements from the range \( [\text{first}, \text{last}) \) as sorted with respect to \( \text{comp} \) and \( \text{proj} \) into the range \( [\text{first}, \text{middle}) \). The rest of the elements in the range \( [\text{middle}, \text{last}) \) are placed in an unspecified order.

Returns: \( \text{last} \) for the overload in namespace \( \text{ranges} \).

Complexity: Approximately \( (\text{last - first}) \times \log(\text{middle - first}) \) comparisons, and twice as many projections.

```cpp
template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr borrowed_iterator_t<R> ranges::partial_sort(R&& r, iterator_t<R> middle,
                                               Comp comp = {}, Proj proj = {});
```

Effects: Equivalent to:

```
return ranges::partial_sort(ranges::begin(r), middle, ranges::end(r), comp, proj);
```

25.8.2.4 partial_sort_copy

```cpp
template<class InputIterator, class RandomAccessIterator>
constexpr RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last);
```
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
 RandomAccessIterator
 partial_sort_copy(ExecutionPolicy&& exec,
     ForwardIterator first, ForwardIterator last,
     RandomAccessIterator result_first,
     RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
 constexpr RandomAccessIterator
 partial_sort_copy(InputIterator first, InputIterator last,
     RandomAccessIterator result_first,
     RandomAccessIterator result_last,
     Compare comp);

template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator, class Compare>
 RandomAccessIterator
 partial_sort_copy(ExecutionPolicy&& exec,
     ForwardIterator first, ForwardIterator last,
     RandomAccessIterator result_first,
     RandomAccessIterator result_last,
     Compare comp);

template<input_iterator I1, sentinel_for<I1> S1, random_access_iterator I2, sentinel_for<I2> S2, class Comp = ranges::less, class Proj1 = identity, class Proj2 = identity>
 requires indirectly_copyable<I1, I2> && sortable<I2, Comp, Proj2> && indirect_strict_weak_order<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
 constexpr ranges::partial_sort_copy_result<I1, I2>
 ranges::partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
     Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, random_access_range R2, class Comp = ranges::less, class Proj1 = identity, class Proj2 = identity>
 requires indirectly_copyable<iterator_t<R1>, iterator_t<R2>> && sortable<iterator_t<R2>, Comp, Proj2> && indirect_strict_weak_order<Comp, projected<iterator_t<R1>, Proj1>, projected<iterator_t<R2>, Proj2>>
 constexpr ranges::partial_sort_copy_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>>
 ranges::partial_sort_copy(R1&& r, R2&& result_r, Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

Let $N$ be $\min(last - first, result_last - result_first)$. Let $\text{comp}$ be $\text{less}()$, and $\text{proj1}$ and $\text{proj2}$ be $\text{identity}()$ for the overloads with no parameters by those names.

**Mandates:** For the overloads in namespace `std`, the expression `*first` is writable (23.3.1) to `result_first`.

**Preconditions:** For the overloads in namespace `std`, `RandomAccessIterator` meets the `Cpp17ValueSwappable` requirements (16.4.4.3), the type of `*result_first` meets the `Cpp17MoveConstructible` (Table 28) and `Cpp17MoveAssignable` (Table 30) requirements.

For iterators $a1$ and $b1$ in `[first, last]$, and iterators $x2$ and $y2$ in `[result_first, result_last]$, after evaluating the assignment `*y2 = *b1`, let $E$ be the value of

$$\text{bool}(\text{invoke}((\text{comp}, \text{invoke}((\text{proj1}, *a1), \text{invoke}((\text{proj2}, *y2))))).$$

Then, after evaluating the assignment `*x2 = *a1`, $E$ is equal to

$$\text{bool}(\text{invoke}((\text{comp}, \text{invoke}((\text{proj2}, *x2), \text{invoke}((\text{proj2}, *y2))))).$$

**Note 1:** Writing a value from the input range into the output range does not affect how it is ordered by $\text{comp}$ and $\text{proj1}$ or $\text{proj2}$. — end note

**Effects:** Places the first $N$ elements as sorted with respect to $\text{comp}$ and $\text{proj2}$ into the range `[result_first, result_first + N]`.

**Returns:**

1. `result_first + N` for the overloads in namespace `std`.
2. `{last, result_first + N}` for the overloads in namespace `ranges`.  

§ 25.8.2.4
Complexity: Approximately \((\text{last} - \text{first}) \times \log N\) comparisons, and twice as many projections.

25.8.2.5 \texttt{is\_sorted} \hfill [is\_sorted]

\texttt{template<class ForwardIterator>}
\texttt{constexpr bool is\_sorted(ForwardIterator first, ForwardIterator last);}\hfill 1

\textit{Effects:} Equivalent to: \texttt{return is\_sorted\_until(first, last) == last};

\texttt{template<class ExecutionPolicy, class ForwardIterator>}
\texttt{bool is\_sorted(ExecutionPolicy&& exec,}
\texttt{ ForwardIterator first, ForwardIterator last);}\hfill 2

\textit{Effects:} Equivalent to:
\texttt{return is\_sorted\_until(std::forward<ExecutionPolicy>(exec), first, last) == last;}

\texttt{template<class ForwardIterator, class Compare>}
\texttt{constexpr bool is\_sorted(ForwardIterator first, ForwardIterator last,}
\texttt{ Compare comp);}\hfill 3

\textit{Effects:} Equivalent to:
\texttt{return is\_sorted\_until(first, last, comp) == last;}

\texttt{template<class ExecutionPolicy, class ForwardIterator, class Compare>}
\texttt{bool is\_sorted(ExecutionPolicy&& exec,}
\texttt{ ForwardIterator first, ForwardIterator last,}
\texttt{ Compare comp);}\hfill 4

\textit{Effects:} Equivalent to:
\texttt{return is\_sorted\_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last;}

\texttt{template<forward_iterator I, sentinel_for<I> S, class Proj = identity,}
\texttt{ indirect\_strict\_weak\_order<projected<I, Proj>> Comp = ranges::less>}
\texttt{constexpr bool ranges::is\_sorted(I first, S last, Comp comp = {}, Proj proj = {});}\hfill 5

\textit{Effects:} Equivalent to:
\texttt{return ranges::is\_sorted\_until(first, last, comp, proj) == last;}

\texttt{template<class ForwardIterator>}
\texttt{constexpr ForwardIterator}
\texttt{is\_sorted\_until(ForwardIterator first, ForwardIterator last);}\hfill 6

\texttt{template<class ExecutionPolicy, class ForwardIterator>}
\texttt{ForwardIterator}
\texttt{is\_sorted\_until(ExecutionPolicy&& exec,}
\texttt{ ForwardIterator first, ForwardIterator last);}\hfill 7

\texttt{template<class ForwardIterator, class Compare>}
\texttt{constexpr ForwardIterator}
\texttt{is\_sorted\_until(ForwardIterator first, ForwardIterator last,}
\texttt{ Compare comp);}\hfill 8

\texttt{template<class ExecutionPolicy, class ForwardIterator, class Compare>}
\texttt{ForwardIterator}
\texttt{is\_sorted\_until(ExecutionPolicy&& exec,}
\texttt{ ForwardIterator first, ForwardIterator last,}
\texttt{ Compare comp);}\hfill 9

\texttt{template<forward_iterator I, sentinel_for<I> S, class Proj = identity,}
\texttt{ indirect\_strict\_weak\_order<projected<I, Proj>> Comp = ranges::less>}
\texttt{constexpr I ranges::is\_sorted\_until(I first, S last, Comp comp = {}, Proj proj = {});}\hfill 10

\textit{Let \texttt{comp} be \texttt{less\{\}} and \texttt{proj} be \texttt{identity\{\}} for the overloads with no parameters by those names.}
Returns: The last iterator \( i \) in \([\text{first}, \text{last}]\) for which the range \([\text{first}, i)\) is sorted with respect to \( \text{comp} \) and \( \text{proj} \).

Complexity: Linear.

25.8.3 Nth element

\[ \text{alg.nth.element} \]

```cpp
template<class RandomAccessIterator>
constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator>
void nth_element(ExecutionPolicy&& exec, RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void nth_element(ExecutionPolicy&& exec, RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp);
```

```cpp
template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less, class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr I
ranges::nth_element(I first, I nth, S last, Comp comp = {}, Proj proj = {});
```

Let \( \text{comp} \) be \( \text{less}() \) and \( \text{proj} \) be \( \text{identity}() \) for the overloads with no parameters by those names.

Preconditions: \([\text{first}, \text{nth})\) and \([\text{nth}, \text{last})\) are valid ranges. For the overloads in namespace \text{std}, \text{RandomAccessIterator} meets the \text{Cpp17ValueSwappable} requirements (16.4.4.3), and the type of \*first meets the \text{Cpp17MoveConstructible} (Table 28) and \text{Cpp17MoveAssignable} (Table 30) requirements.

Effects: After \text{nth_element} the element in the position pointed to by \text{nth} is the element that would be in that position if the whole range were sorted with respect to \text{comp} and \text{proj}, unless \text{nth} == \text{last}. Also for every iterator \( i \) in the range \([\text{first}, \text{nth})\) and every iterator \( j \) in the range \([\text{nth}, \text{last})\) it holds that: \( \text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, \*j), \text{invoke}(\text{proj}, \*i))) \) is \( \text{false} \).

Returns: last for the overload in namespace \text{ranges}.

Complexity: For the overloads with no \text{ExecutionPolicy}, linear on average. For the overloads with an \text{ExecutionPolicy}, \( \Theta(N) \) applications of the predicate, and \( \Theta(N \log N) \) swaps, where \( N = \text{last} - \text{first} \).

```cpp
template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr borrowed_iterator_t<R>
ranges::nth_element(R&& r, iterator_t<R> nth, Comp comp = {}, Proj proj = {});
```

Effects: Equivalent to:
```cpp
return ranges::nth_element(ranges::begin(r), nth, ranges::end(r), comp, proj);
```

25.8.4 Binary search

25.8.4.1 General

All of the algorithms in 25.8.4 are versions of binary search and assume that the sequence being searched is partitioned with respect to an expression formed by binding the search key to an argument of the comparison function. They work on non-random access iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access iterators, because these algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a linear number of steps.
25.8.4.2  lower_bound

```cpp
template<class ForwardIterator, class T>
constexpr ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
    const T& value);
```

```cpp
template<class ForwardIterator, class T, class Compare>
constexpr ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp);
```

```cpp
template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity,
    indirect_strict_weak_order<const T*, projected<I, Proj>> Comp = ranges::less>
constexpr I ranges::lower_bound(I first, S last, const T& value, Comp comp = {},
    Proj proj = {});
```

```cpp
template<forward_range R, class T, class Proj = identity,
    indirect_strict_weak_order<const T*, projected<iterator_t<R>, Proj>> Comp =
    ranges::less>
constexpr borrowed_iterator_t<R> ranges::lower_bound(R&& r, const T& value, Comp comp = {}, Proj proj = {});
```

1. Let `comp` be `less{}` and `proj` be `identity{}` for overloads with no parameters by those names.

2. **Preconditions:** The elements `e` of `[first, last)` are partitioned with respect to the expression `bool(invoke(comp, invoke(proj, e), value))`.  

3. **Returns:** The furthermost iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, i)` `bool(invoke(comp, invoke(proj, *j), value))` is true.

4. **Complexity:** At most `log_2(last - first) + O(1)` comparisons and projections.

25.8.4.3  upper_bound

```cpp
template<class ForwardIterator, class T>
constexpr ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
    const T& value);
```

```cpp
template<class ForwardIterator, class T, class Compare>
constexpr ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp);
```

```cpp
template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity,
    indirect_strict_weak_order<const T*, projected<I, Proj>> Comp = ranges::less>
constexpr I ranges::upper_bound(I first, S last, const T& value, Comp comp = {},
    Proj proj = {});
```

```cpp
template<forward_range R, class T, class Proj = identity,
    indirect_strict_weak_order<const T*, projected<iterator_t<R>, Proj>> Comp =
    ranges::less>
constexpr borrowed_iterator_t<R> ranges::upper_bound(R&& r, const T& value, Comp comp = {}, Proj proj = {});
```

1. Let `comp` be `less{}` and `proj` be `identity{}` for overloads with no parameters by those names.

2. **Preconditions:** The elements `e` of `[first, last)` are partitioned with respect to the expression `!bool(invoke(comp, value, invoke(proj, e)))`.  

3. **Returns:** The furthermost iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, i), !bool(invoke(comp, value, invoke(proj, *j)))` is true.

4. **Complexity:** At most `log_2(last - first) + O(1)` comparisons and projections.

25.8.4.4  equal_range

```cpp
template<class ForwardIterator, class T>
constexpr pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first,
    ForwardIterator last, const T& value);
```
template<class ForwardIterator, class T, class Compare>
constexpr pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity,
indirect_strict_weak_order<const T*, projected<I, Proj>> Comp = ranges::less>
constexpr subrange<I>
ranges::equal_range(I first, S last, const T& value, Comp comp = {}, Proj proj = {});

template<forward_range R, class T, class Proj = identity,
indirect_strict_weak_order<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less>
constexpr borrowed_subrange_t<R>
ranges::equal_range(R&& r, const T& value, Comp comp = {}, Proj proj = {});

1 Let comp be less{} and proj be identity{} for overloads with no parameters by those names.
2 Preconditions: The elements e of [first, last) are partitioned with respect to the expressions
bool(invoke(comp, invoke(proj, e), value)) and !bool(invoke(comp, value, invoke(proj, e))). Also, for all elements e of [first, last), bool(comp(e, value)) implies !bool(comp(value, e)) for the overloads in namespace std.
3 Returns:
   (3.1) — For the overloads in namespace std:
   \{lower_bound(first, last, value, comp),
   upper_bound(first, last, value, comp)\}
   (3.2) — For the overloads in namespace ranges:
   \{ranges::lower_bound(first, last, value, comp, proj),
   ranges::upper_bound(first, last, value, comp, proj)\}
4 Complexity: At most 2 * log₂(last - first) + O(1) comparisons and projections.

25.8.4.5 binary_search

template<class ForwardIterator, class T>
constexpr bool
binary_search(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
constexpr bool
binary_search(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity,
indirect_strict_weak_order<const T*, projected<I, Proj>> Comp = ranges::less>
constexpr bool ranges::binary_search(I first, S last, const T& value, Comp comp = {}, Proj proj = {});

template<forward_range R, class T, class Proj = identity,
indirect_strict_weak_order<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less>
constexpr bool ranges::binary_search(R&& r, const T& value, Comp comp = {}, Proj proj = {});

1 Let comp be less{} and proj be identity{} for overloads with no parameters by those names.
2 Preconditions: The elements e of [first, last) are partitioned with respect to the expressions
bool(invoke(comp, invoke(proj, e), value)) and !bool(invoke(comp, value, invoke(proj, e))). Also, for all elements e of [first, last), bool(comp(e, value)) implies !bool(comp(value, e)) for the overloads in namespace std.
3 Returns: true if and only if for some iterator i in the range [first, last), !bool(invoke(comp, invoke(proj, *i), value)) && !bool(invoke(comp, value, invoke(proj, *i))) is true.
4 Complexity: At most log₂(last - first) + Θ(1) comparisons and projections.
25.8.5 Partitions

\[ \text{is\_partioned} \]

\[
\begin{align*}
\text{template<class InputIterator, class Predicate>} & \quad \text{constexpr bool is\_partitioned(InputIterator first, InputIterator last, Predicate pred);} \\
\text{template<class ExecutionPolicy, class ForwardIterator, class Predicate>} & \quad \text{bool is\_partitioned(ExecutionPolicy\&\& exec,}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{ForwardIterator first, ForwardIterator last, Predicate pred);} \\
\text{template<input\_iterator I, sentinel\_for<\!\!I\!\!> S, class Proj = identity,}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{indirect\_unary\_predicate<projected<\!\!I\!\!, Proj>> Pred>} \\
\text{constexpr bool ranges::is\_partitioned(I first, S last, Pred pred, Proj proj = {});} \\
\text{template<input\_range R, class Proj = identity,}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{indirect\_unary\_predicate<projected<iterator\_t<R>, Proj>> Pred>} \\
\text{constexpr bool ranges::is\_partitioned(R&& r, Pred pred, Proj proj = {});} \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{Let proj be identity{} for the overloads with no parameter named proj.} \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{Returns: true if and only if the elements e of } [\text{first, last}) \text{ are partitioned with respect to the } \text{expression bool(invoke(pred, invoke(proj, e)))}. \\
& \quad \text{Complexity: Linear. At most last - first applications of pred and proj.} \\
\end{align*}
\]

\[
\begin{align*}
\text{template<class ForwardIterator, class Predicate>} & \quad \text{constexpr ForwardIterator} \\
\text{partition(ForwardIterator first, ForwardIterator last, Predicate pred);} \\
\text{template<class ExecutionPolicy, class ForwardIterator, class Predicate>} & \quad \text{ForwardIterator} \\
\text{partition(ExecutionPolicy\&\& exec,} \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{ForwardIterator first, ForwardIterator last, Predicate pred);} \\
\text{template<permutable I, sentinel\_for<\!\!I\!\!> S, class Proj = identity,}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{indirect\_unary\_predicate<projected<\!\!I\!\!, Proj>> Pred>} \\
\text{constexpr subrange<I>} \\
\end{align*}
\]

\[
\begin{align*}
\text{ranges::partition(I first, S last, Pred pred, Proj proj = {});} \\
\text{template<forward\_range R, class Proj = identity,}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{indirect\_unary\_predicate<projected<iterator\_t<R>, Proj>> Pred> requires permutable<iterator\_t<R>)} \\
\text{constexpr borrowed\_subrange\_t<R>} \\
\end{align*}
\]

\[
\begin{align*}
\text{ranges::partition(R&& r, Pred pred, Proj proj = {});} \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{Let proj be identity{} for the overloads with no parameter named proj and let } E(x) \text{ be bool(invoke(pred, invoke(proj, x)))}. \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{Preconditions: For the overloads in namespace std, ForwardIterator meets the Cpp17ValueSwappable requirements (16.4.4.3).} \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{Effects: Places all the elements e in } [\text{first, last}) \text{ that satisfy } E(e) \text{ before all the elements that do not.} \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{Returns: Let } i \text{ be an iterator such that } E(*j) \text{ is true for every iterator } j \text{ in } [\text{first, i}) \text{ and false for every iterator } j \text{ in } [i, \text{last}) \text{. Returns:} \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{(7.1) i for the overloads in namespace std.} \\
& \quad \text{(7.2) } \{i, \text{last}) \text{ for the overloads in namespace ranges.} \\
\end{align*}
\]

\[
\begin{align*}
\text{template<class BidirectionalIterator, class Predicate>} & \quad \text{BidirectionalIterator} \\
\text{stable\_partition(BidirectionalIterator first, BidirectionalIterator last, Predicate pred);} \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{Complexity: For the overload with no ExecutionPolicy, exactly N applications of the predicate and projection.} \\
& \quad \text{At most N/2 swaps if the type of first meets the Cpp17BidirectionalIterator requirements for the overloads in namespace std or models bidirectional\_iterator for the overloads in namespace ranges, and at most N swaps otherwise.} \\
\text{(8.2) For the overload with an ExecutionPolicy, } O(N \log N) \text{ swaps and } O(N) \text{ applications of the predicate.} \\
\end{align*}
\]

\[
\begin{align*}
\text{template<class BidirectionalIterator, class Predicate>} & \quad \text{BidirectionalIterator} \\
\text{stable\_partition(BidirectionalIterator first, BidirectionalIterator last, Predicate pred);} \\
\end{align*}
\]
template<class ExecutionPolicy, class BidirectionalIterator, class Predicate>
    BidirectionalIterator
    stable_partition(ExecutionPolicy&& exec,
        BidirectionalIterator first, BidirectionalIterator last, Predicate pred);

template<bidirectional_iterator I, sentinel_for<I> S, class Proj = identity,
    indirect_unary_predicate<projected<I, Proj>> Pred>
requires permutable<I>
    subrange<I> ranges::stable_partition(I first, S last, Pred pred, Proj proj = {});

template<bidirectional_range R, class Proj = identity,
    indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
requires permutable<iterator_t<R>>
borrowed_subrange_t<R> ranges::stable_partition(R&& r, Pred pred, Proj proj = {});

Let proj be identity{} for the overloads with no parameter named proj and let \( E(x) \) be bool(invoke(pred, invoke(proj, x))).

Preconditions: For the overloads in namespace std, BidirectionalIterator meets the Cpp17Value-Swappable requirements (16.4.4.3) and the type of *first meets the Cpp17MoveConstructible (Table 28) and Cpp17MoveAssignable (Table 30) requirements.

Effects: Places all the elements \( e \) in \([first, last)\) that satisfy \( E(e) \) before all the elements that do not. The relative order of the elements in both groups is preserved.

Returns: Let \( i \) be an iterator such that for every iterator \( j \) in \([first, i)\), \( E(*j) \) is true, and for every iterator \( j \) in the range \([i, last)\), \( E(*j) \) is false. Returns:

(12.1) \( i \) for the overloads in namespace std.
(12.2) \( \{i, last\} \) for the overloads in namespace ranges.

Complexity: Let \( N = last - first \):

(13.1) For the overloads with no ExecutionPolicy, at most \( N \log N \) swaps, but only \( \Theta(N) \) swaps if there is enough extra memory. Exactly \( N \) applications of the predicate and projection.
(13.2) For the overload with an ExecutionPolicy, \( \Theta(N \log N) \) swaps and \( \Theta(N) \) applications of the predicate.

Let proj be identity{} for the overloads with no parameter named proj and let \( E(x) \) be bool(invoke(pred, invoke(proj, x))).

Mandates: For the overloads in namespace std, the expression *first is writable (23.3.1) to out_true and out_false.
Preconditions: The input range and output ranges do not overlap.

[Note 1: For the overload with an ExecutionPolicy, there might be a performance cost if \texttt{first}'s value type does not meet the \texttt{Cpp17CopyConstructible} requirements. — end note]

Effects: For each iterator \(i\) in \([\texttt{first}, \texttt{last})\), copies \(*i\) to the output range beginning with \texttt{out\_true} if \(\texttt{E}(*i)\) is \texttt{true}, or to the output range beginning with \texttt{out\_false} otherwise.

Returns: Let \(o1\) be the end of the output range beginning at \texttt{out\_true}, and \(o2\) the end of the output range beginning at \texttt{out\_false}. Returns

\begin{align*}
& - \{o1, o2\} \text{ for the overloads in namespace std.} \\
& - \{\texttt{last, o1, o2}\} \text{ for the overloads in namespace ranges.}
\end{align*}

Complexity: Exactly \(\texttt{last} - \texttt{first}\) applications of \texttt{pred} and \texttt{proj}.

\begin{verbatim}
template<class ForwardIterator, class Predicate>
constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last, Predicate pred);

template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
    indirect_unary_predicate<projected<I, Proj>> Pred>
constexpr I ranges::partition_point(I first, S last, Pred pred, Proj proj = {});

template<forward_range R, class Proj = identity,
    indirect_unary_predicate<iterator_t<R>, Proj>> Pred>
constexpr borrowed_iterator_t<R>
    ranges::partition_point(R&& r, Pred pred, Proj proj = {});
\end{verbatim}

Preconditions: The elements \(e\) of \([\texttt{first}, \texttt{last})\) are partitioned with respect to \(\texttt{E}(e)\).

Returns: An iterator \(\texttt{mid}\) such that \(\texttt{E}(*i)\) is \texttt{true} for all iterators \(i\) in \([\texttt{first}, \texttt{mid})\), and \texttt{false} for all iterators \(i\) in \([\texttt{mid}, \texttt{last})\).

Complexity: \(\Theta(\log(\texttt{last} - \texttt{first}))\) applications of \texttt{pred} and \texttt{proj}.

\subsection*{25.8.6 Merge \hfill [alg.merge]}

\begin{verbatim}
template<class InputIterator1, class InputIterator2,
    class OutputIterator>
constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class ForwardIterator>
ForwardIterator
    merge(ExecutionPolicy&& exec,
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        ForwardIterator result);

template<class InputIterator1, class InputIterator2,
    class OutputIterator, class Compare>
constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result, Compare comp);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class ForwardIterator, class Compare>
ForwardIterator
    merge(ExecutionPolicy&& exec,
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        ForwardIterator result, Compare comp);
\end{verbatim}

\section*{§ 25.8.6}
template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2, weakly_incrementable O, class Comp = ranges::less, class Proj1 = identity, class Proj2 = identity>
requires mergeable<I1, I2, O, Comp, Proj1, Proj2>
constexpr ranges::merge_result<I1, I2, O>
ranges::merge(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

Let \(N\) be \((last1 - first1) + (last2 - first2)\). Let \(\text{comp}\) be \(\text{less}\{}\), \(\text{proj1}\) be \(\text{identity}\{}\), and \(\text{proj2}\) be \(\text{identity}\{}\), for the overloads with no parameters by those names.

Preconditions: The ranges \([first1, last1)\) and \([first2, last2)\) are sorted with respect to \(\text{comp}\) and \(\text{proj1}\) or \(\text{proj2}\), respectively. The resulting range does not overlap with either of the original ranges.

Effects: Copies all the elements of the two ranges \([first1, last1)\) and \([first2, last2)\) into the range \([result, result_last)\), where \(result_last\) is \(result + N\). If an element \(a\) precedes \(b\) in an input range, \(a\) is copied into the output range before \(b\). If \(e1\) is an element of \([first1, last1)\) and \(e2\) of \([first2, last2)\), \(e2\) is copied into the output range before \(e1\) if and only if \(\text{bool(invoke(\text{comp}, invoke(\text{proj2}, e2), invoke(\text{proj1}, e1))) is true}\). Returns:

1. \(\text{result_last}\) for the overloads in namespace \(\text{std}\).
2. \{last1, last2, result_last\} for the overloads in namespace \(\text{ranges}\).

Complexity:

1. For the overloads with no \(\text{ExecutionPolicy}\), at most \(N - 1\) comparisons and applications of each projection.
2. For the overloads with an \(\text{ExecutionPolicy}\), \(\mathcal{O}(N)\) comparisons.

Remarks: Stable (16.4.6.8).

template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last);

template<class ExecutionPolicy, class BidirectionalIterator>
void inplace_merge(ExecutionPolicy&& exec, BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last, Compare comp);

template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
void inplace_merge(ExecutionPolicy&& exec, BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last, Compare comp);

Let \(\text{comp}\) be \(\text{less}\{}\) and \(\text{proj}\) be \(\text{identity}\{}\) for the overloads with no parameters by those names.

§ 25.8.6

1126
Preconditions: \([\text{first, middle})\) and \([\text{middle, last})\) are valid ranges sorted with respect to \(\text{comp}\) and \(\text{proj}\). For the overloads in namespace std, BidirectionalIterator meets the Cpp17ValueSwapable requirements (16.4.4.3) and the type of \(*\text{first}\) meets the Cpp17MoveConstructible (Table 28) and Cpp17MoveAssignable (Table 30) requirements.

Effects: Merges two sorted consecutive ranges \([\text{first, middle})\) and \([\text{middle, last})\), putting the result of the merge into the range \([\text{first, last})\). The resulting range is sorted with respect to \(\text{comp}\) and \(\text{proj}\).

Returns: \(\text{last}\) for the overload in namespace ranges.

Complexity: Let \(N = \text{last} - \text{first}\):

1. For the overloads with no ExecutionPolicy, and if enough additional memory is available, exactly \(N - 1\) comparisons.
2. Otherwise, \(\Theta(N \log N)\) comparisons.

In either case, twice as many projections as comparisons.

Remarks: Stable (16.4.6.8).

template<bidirectional_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
borrowed_iterator_t<R>
ranges::inplace_merge(R&& r, iterator_t<R> middle, Comp comp = {}, Proj proj = {});

Effects: Equivalent to:

\[
\text{return ranges::inplace_merge(ranges::begin(r), middle, ranges::end(r), comp, proj);};
\]

25.8.7 Set operations on sorted structures [alg.set.operations]

25.8.7.1 General [alg.set.operations.general]

Subclause 25.8.7 defines all the basic set operations on sorted structures. They also work with multisets (22.4.7) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to multisets in a standard way by defining setUnion to contain the maximum number of occurrences of every element, setIntersection to contain the minimum, and so on.

25.8.7.2 includes [includes]

template<class InputIterator1, class InputIterator2>
constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
bool includes(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
Compare comp);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Compare>
bool includes(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
Compare comp);

template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,>
class Proj1 = identity, class Proj2 = identity,
indirect_strict_weak_order<projected<I1, Proj1>,
projected<I2, Proj2>> Comp = ranges::less>
constexpr bool ranges::includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = {},
Proj1 proj1 = {}, Proj2 proj2 = {});
template<input_range R1, input_range R2, class Proj1 = identity,  
class Proj2 = identity,  
indirect_strict_weak_order<
projected<
iterator_t<R1>, Proj1>,  
projected<
iterator_t<R2>, Proj2>> Comp = ranges::less>
constexpr bool ranges::includes(R1&& r1, R2&& r2, Comp comp = {},  
Proj1 proj1 = {}, Proj2 proj2 = {});

Let \texttt{comp} be \texttt{less{}}, \texttt{proj1} be \texttt{identity{}}, and \texttt{proj2} be \texttt{identity{}}, for the overloads with no parameters by those names.

Preconditions: The ranges \texttt{[first1, last1)} and \texttt{[first2, last2)} are sorted with respect to \texttt{comp} and \texttt{proj1} or \texttt{proj2}, respectively.

Returns: \texttt{true} if and only if \texttt{[first2, last2)} is a subsequence of \texttt{[first1, last1)}.

\textit{Note 1:} A sequence \textit{S} is a subsequence of another sequence \textit{T} if \textit{S} can be obtained from \textit{T} by removing some, all, or none of \textit{T}'s elements and keeping the remaining elements in the same order. —end note

Complexity: At most \texttt{2 \ast ((last1 - first1) + (last2 - first2)) - 1} comparisons and applications of each projection.

25.8.7.3 \texttt{set\_union}

\begin{verbatim}
template<class InputIterator1, class InputIterator2, class OutputIterator>
constexpr OutputIterator
set\_union(InputIterator1 first1, InputIterator1 last1,  
InputIterator2 first2, InputIterator2 last2,  
OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,  
class ForwardIterator>
ForwardIterator
set\_union(ExecutionPolicy&& exec,  
ForwardIterator1 first1, ForwardIterator1 last1,  
ForwardIterator2 first2, ForwardIterator2 last2,  
ForwardIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
constexpr OutputIterator
set\_union(InputIterator1 first1, InputIterator1 last1,  
InputIterator2 first2, InputIterator2 last2,  
OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,  
class ForwardIterator, class Compare>
ForwardIterator
set\_union(ExecutionPolicy&& exec,  
ForwardIterator1 first1, ForwardIterator1 last1,  
ForwardIterator2 first2, ForwardIterator2 last2,  
ForwardIterator result, Compare comp);

template<input\_iterator I1, sentinel\_for<I1> S1, input\_iterator I2, sentinel\_for<I2> S2,  
weakly\_incrementable O, class Compare = ranges::less,  
class Proj1 = identity, class Proj2 = identity>
requires mergeable<I1, I2, O, Comp, Proj1, Proj2>
constexpr ranges::set\_union_result<I1, I2, O>
ranges::set\_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Compare comp = {},  
Proj1 proj1 = {}, Proj2 proj2 = {});

template<input\_range R1, input\_range R2, weakly\_incrementable O,  
class Compare = ranges::less, class Proj1 = identity, class Proj2 = identity>
requires mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
constexpr ranges::set\_union_result<borrowed\_iterator_t<R1>, borrowed\_iterator_t<R2>, O>
ranges::set\_union(R1&& r1, R2&& r2, O result, Compare comp = {},  
Proj1 proj1 = {}, Proj2 proj2 = {});
\end{verbatim}

Let \texttt{comp} be \texttt{less{}}, and \texttt{proj1} and \texttt{proj2} be \texttt{identity{}} for the overloads with no parameters by those names.

§ 25.8.7.3

1128
Preconditions: The ranges \([\text{first}_1, \text{last}_1)\) and \([\text{first}_2, \text{last}_2)\) are sorted with respect to \(\text{comp}\) and \(\text{proj}_1\) or \(\text{proj}_2\), respectively. The resulting range does not overlap with either of the original ranges.

Effects: Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

Returns: Let \(\text{result\_last}\) be the end of the constructed range. Returns

- \(\text{result\_last}\) for the overloads in namespace \(\text{std}\).
- \([\text{last}_1, \text{last}_2, \text{result\_last}]\) for the overloads in namespace \(\text{ranges}\).

Complexity: At most \(2 \times ((\text{last}_1 - \text{first}_1) + (\text{last}_2 - \text{first}_2)) - 1\) comparisons and applications of each projection.

Remarks: Stable \((16.4.6.8)\). If \([\text{first}_1, \text{last}_1)\) contains \(m\) elements that are equivalent to each other and \([\text{first}_2, \text{last}_2)\) contains \(n\) elements that are equivalent to them, then all \(m\) elements from the first range are copied to the output range, in order, and then the final \(\max(n - m, 0)\) elements from the second range are copied to the output range, in order.

25.8.7.4 \(\text{set}\_\text{intersection}\)
Preconditions: The ranges \([\text{first}_1, \text{last}_1)\) and \([\text{first}_2, \text{last}_2)\) are sorted with respect to \(\text{comp}\) and \(\text{proj}_1\) or \(\text{proj}_2\), respectively. The resulting range does not overlap with either of the original ranges.

Effects: Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

Returns: Let \(\text{result}_\text{last}\) be the end of the constructed range. Returns

\[
\begin{align*}
\text{— result}_\text{last} & \text{ for the overloads in namespace } \text{std}. \\
\text{— } & \{\text{last}_1, \text{last}_2, \text{result}_\text{last}\} \text{ for the overloads in namespace } \text{ranges}.
\end{align*}
\]

Complexity: At most \(2 \times ((\text{last}_1 - \text{first}_1) + (\text{last}_2 - \text{first}_2)) - 1\) comparisons and applications of each projection.

Remarks: Stable (16.4.6.8). If \([\text{first}_1, \text{last}_1)\) contains \(m\) elements that are equivalent to each other and \([\text{first}_2, \text{last}_2)\) contains \(n\) elements that are equivalent to them, the first \(\min(m, n)\) elements are copied from the first range to the output range, in order.

25.8.7.5 \textbf{set_difference}

\[
\begin{align*}
\text{template}\langle\text{class } & \text{InputIterator}_1, \text{class } \text{InputIterator}_2, \\
& \text{class } \text{OutputIterator}\rangle \quad \text{constexpr } \text{OutputIterator} \\
& \text{set_difference}(\text{InputIterator}_1 \text{ first}_1, \text{InputIterator}_1 \text{ last}_1, \\
& \text{InputIterator}_2 \text{ first}_2, \text{InputIterator}_2 \text{ last}_2, \text{OutputIterator} \text{ result}); \\
\text{template}\langle\text{class } & \text{ExecutionPolicy}, \text{class } \text{ForwardIterator}_1, \text{class } \text{ForwardIterator}_2, \\
& \text{class } \text{ForwardIterator}\rangle \quad \text{ForwardIterator} \\
& \text{set_difference}(\text{ExecutionPolicy}&& \text{exec}, \\
& \text{ForwardIterator}_1 \text{ first}_1, \text{ForwardIterator}_1 \text{ last}_1, \\
& \text{ForwardIterator}_2 \text{ first}_2, \text{ForwardIterator}_2 \text{ last}_2, \text{ForwardIterator} \text{ result}); \\
\text{template}\langle\text{class } & \text{InputIterator}_1, \text{class } \text{InputIterator}_2, \\
& \text{class } \text{Compare}\rangle \quad \text{constexpr } \text{OutputIterator} \\
& \text{set_difference}(\text{InputIterator}_1 \text{ first}_1, \text{InputIterator}_1 \text{ last}_1, \\
& \text{InputIterator}_2 \text{ first}_2, \text{InputIterator}_2 \text{ last}_2, \text{OutputIterator} \text{ result}, \text{Compare} \text{ comp}); \\
\text{template}\langle\text{class } & \text{ExecutionPolicy}, \text{class } \text{ForwardIterator}_1, \text{class } \text{ForwardIterator}_2, \\
& \text{class } \text{Compare}\rangle \quad \text{ForwardIterator} \\
& \text{set_difference}(\text{ExecutionPolicy}&& \text{exec}, \\
& \text{ForwardIterator}_1 \text{ first}_1, \text{ForwardIterator}_1 \text{ last}_1, \\
& \text{ForwardIterator}_2 \text{ first}_2, \text{ForwardIterator}_2 \text{ last}_2, \text{ForwardIterator} \text{ result}, \text{Compare} \text{ comp}); \\
\text{template}\langle\text{input_iterator } & \text{I}_1, \text{sentinel_for}<\text{I}_1> \text{ S}_1, \text{input_iterator } \text{I}_2, \text{sentinel_for}<\text{I}_2> \text{ S}_2, \\
& \text{weakly_incrementable } 0, \text{class } \text{Compare} = \text{ranges}::\text{less}, \\
& \text{class } \text{Proj}_1 = \text{identity}, \text{class } \text{Proj}_2 = \text{identity}\rangle \quad \text{requires } \text{mergeable}<\text{I}_1, \text{I}_2, 0, \text{Comp}, \text{Proj}_1, \text{Proj}_2> \\
& \text{constexpr ranges}::\text{set_difference_result}<\text{I}_1, 0> \\
& \text{ranges}::\text{set_difference}(\text{I}_1 \text{ first}_1, \text{I}_1 \text{ last}_1, \text{I}_2 \text{ first}_2, \text{I}_2 \text{ last}_2, 0, \text{result}, \\
& \text{Comp} \text{ comp} = \emptyset, \text{Proj}_1 \text{ proj}_1 = \emptyset, \text{Proj}_2 \text{ proj}_2 = \emptyset); \\
\text{template}\langle\text{input_range } & \text{R}_1, \text{input_range } \text{R}_2, \text{weakly_incrementable } 0, \\
& \text{class } \text{Comp} = \text{ranges}::\text{less}, \text{class } \text{Proj}_1 = \text{identity}, \text{class } \text{Proj}_2 = \text{identity}\rangle \quad \text{requires } \text{mergeable}<\text{iterator}_t<\text{R}_1>, \text{iterator}_t<\text{R}_2>, 0, \text{Comp}, \text{Proj}_1, \text{Proj}_2> \\
& \text{constexpr ranges}::\text{set_difference_result}<\text{borrowed_iterator}_t<\text{R}_1>, 0> \\
& \text{ranges}::\text{set_difference}(\text{R}_1 \text{ first}_1, \text{R}_2 \text{ first}_2, \text{R}_1 \text{ last}_2, \text{R}_2 \text{ last}_2, 0, \text{result}, \\
& \text{Comp} \text{ comp} = \emptyset, \text{Proj}_1 \text{ proj}_1 = \emptyset, \text{Proj}_2 \text{ proj}_2 = \emptyset); \\
\end{align*}
\]

Let \(\text{comp}\) be \(\text{less}\{\}\), and \(\text{proj}_1\) and \(\text{proj}_2\) be \(\text{identity}\{\}\) for the overloads with no parameters by those names.
Preconditions: The ranges \([\text{first}_1, \text{last}_1)\) and \([\text{first}_2, \text{last}_2)\) are sorted with respect to \(\text{comp}\) and \(\text{proj}_1\) or \(\text{proj}_2\), respectively. The resulting range does not overlap with either of the original ranges.

Effects: Copies the elements of the range \([\text{first}_1, \text{last}_1)\) which are not present in the range \([\text{first}_2, \text{last}_2)\) to the range beginning at \(\text{result}\). The elements in the constructed range are sorted.

Returns: Let \(\text{result}_\text{last}\) be the end of the constructed range. Returns

- \(\text{result}_\text{last}\) for the overloads in namespace std.
- \(\{\text{last}_1, \text{result}_\text{last}\}\) for the overloads in namespace ranges.

Complexity: At most \(2 \times (\text{last}_1 - \text{first}_1) + (\text{last}_2 - \text{first}_2)\) - 1 comparisons and applications of each projection.

Remarks: If \([\text{first}_1, \text{last}_1)\) contains \(m\) elements that are equivalent to each other and \([\text{first}_2, \text{last}_2)\) contains \(n\) elements that are equivalent to them, the last \(\max(m - n, 0)\) elements from \([\text{first}_1, \text{last}_1)\) is copied to the output range, in order.

25.8.7.6 \text{set\_symmetric\_difference} \hfill \text{[set.symmetric.difference]}

\begin{verbatim}
template<class InputIterator1, class InputIterator2,  
class OutputIterator>
constexpr OutputIterator  
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,  
InputIterator2 first2, InputIterator2 last2,  
OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,  
class ForwardIterator>
ForwardIterator  
set_symmetric_difference(ExecutionPolicy&& exec,  
ForwardIterator1 first1, ForwardIterator1 last1,  
ForwardIterator2 first2, ForwardIterator2 last2,  
ForwardIterator result);

template<class InputIterator1, class InputIterator2,  
class OutputIterator, class Compare>
constexpr OutputIterator  
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,  
InputIterator2 first2, InputIterator2 last2,  
OutputIterator result, Compare comp);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,  
class ForwardIterator, class Compare>
ForwardIterator  
set_symmetric_difference(ExecutionPolicy&& exec,  
ForwardIterator1 first1, ForwardIterator1 last1,  
ForwardIterator2 first2, ForwardIterator2 last2,  
ForwardIterator result, Compare comp);

template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,  
weakly_incrementable O, class Comp = ranges::less,  
class Proj1 = identity, class Proj2 = identity>
requires mergeable<I1, I2, O, Comp, Proj1, Proj2>
constexpr ranges::set_symmetric_difference_result<I1, I2, O>  
ranges::set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,  
Comp comp = {}, Proj1 proj1 = {},  
Proj2 proj2 = {});

template<input_range R1, input_range R2, weakly_incrementable O,  
class Comp = ranges::less, class Proj1 = identity, class Proj2 = identity>
requires mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
constexpr ranges::set_symmetric_difference_result<borrowed_iterator_t<R1>,  
borrowed_iterator_t<R2>, O>  
ranges::set_symmetric_difference(R1&& r1, R2&& r2, O result, Comp comp = {}),
\end{verbatim}
Proj1 proj1 = {}, Proj2 proj2 = {};

1 Let \( \text{comp} \) be \text{less}\{}{} and \( \text{proj1} \) and \( \text{proj2} \) be \text{identity}\{}{} for the overloads with no parameters by those names.  

2 \text{Preconditions:} The ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) are sorted with respect to \( \text{comp} \) and \( \text{proj1} \) or \( \text{proj2} \), respectively. The resulting range does not overlap with either of the original ranges.

3 \text{Effects:} Copies the elements of the range \([\text{first1}, \text{last1})\) that are not present in the range \([\text{first2}, \text{last2})\), and the elements of the range \([\text{first2}, \text{last2})\) that are not present in the range \([\text{first1}, \text{last1})\) to the range beginning at \( \text{result} \). The elements in the constructed range are sorted.

4 \text{Returns:} Let \( \text{result\_last} \) be the end of the constructed range. Returns

\begin{align*}
(4.1) & \quad \text{result\_last for the overloads in namespace std.} \\
(4.2) & \quad \{\text{last1, last2, result\_last}\} \text{ for the overloads in namespace ranges.}
\end{align*}

5 \text{Complexity:} At most \( 2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1 \) comparisons and applications of each projection.

6 \text{Remarks:} Stable (16.4.6.8). If \([\text{first1}, \text{last1})\) contains \( m \) elements that are equivalent to each other and \([\text{first2}, \text{last2})\) contains \( n \) elements that are equivalent to them, then \( |m - n| \) of those elements shall be copied to the output range: the last \( m - n \) of these elements from \([\text{first1}, \text{last1})\) if \( m > n \), and the last \( n - m \) of these elements from \([\text{first2}, \text{last2})\) if \( m < n \). In either case, the elements are copied in order.

25.8.8 Heap operations \[\text{alg.heap.operations}\]

25.8.8.1 General \[\text{alg.heap.operations.general}\]

1 A random access range \([a, b)\) is a heap with respect to \( \text{comp} \) and \( \text{proj} \) for a comparator and projection \( \text{comp} \) and \( \text{proj} \) if its elements are organized such that:

\begin{align*}
(1.1) & \quad \text{With } N = b - a, \text{ for all } i, 0 < i < N, \text{ bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, a[\lfloor i/2 \rfloor])), \text{invoke}(\text{proj}, a[i]))) \text{ is false.} \\
(1.2) & \quad *a \text{ may be removed by } \text{pop\_heap}, \text{ or a new element added by } \text{push\_heap}, \text{ in } \mathcal{O}(\log N) \text{ time.}
\end{align*}

2 These properties make heaps useful as priority queues.

3 \text{make\_heap} converts a range into a heap and \text{sort\_heap} turns a heap into a sorted sequence.

25.8.8.2 push\_heap \[\text{push.heap}\]

\begin{verbatim}
template<class RandomAccessIterator>
constexpr void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
constexpr void push_heap(RandomAccessIterator first, RandomAccessIterator last, 
                        Compare comp);

template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less, 
          class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr I
ranges::push_heap(I first, S last, Comp comp = {}, Proj proj = {});

template<random_access_iterator R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr borrowed_iterator_t<R>
ranges::push_heap(R&& r, Comp comp = {}, Proj proj = {});
\end{verbatim}

1 Let \( \text{comp} \) be \text{less}\{}{} and \( \text{proj} \) be \text{identity}\{}{} for the overloads with no parameters by those names.

2 \text{Preconditions:} The range \([\text{first}, \text{last} - 1)\) is a valid heap with respect to \( \text{comp} \) and \( \text{proj} \). For the overloads in namespace std, the type of \*\text{first} meets the Cpp17MoveConstructible requirements (Table 28) and the Cpp17MoveAssignble requirements (Table 30).

3 \text{Effects:} Places the value in the location \( \text{last} - 1 \) into the resulting heap \([\text{first}, \text{last})\).

4 \text{Returns:} \( \text{last} \) for the overloads in namespace ranges.
Complexity: At most $\log(last - first)$ comparisons and twice as many projections.

25.8.8.3 pop_heap

```cpp
template<class RandomAccessIterator>
constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                        Compare comp);
```

```cpp
template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
         class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr I
ranges::pop_heap(I first, S last, Comp comp = {}, Proj proj = {});
```

```cpp
template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr borrowed_iterator_t<R>
ranges::pop_heap(R&& r, Comp comp = {}, Proj proj = {});
```

Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

Preconditions: The range `[first, last)` is a valid non-empty heap with respect to `comp` and `proj`. For the overloads in namespace `std`, `RandomAccessIterator` meets the `Cpp17ValueSwappable` requirements (16.4.4.3) and the type of `*first` meets the `Cpp17MoveConstructible` (Table 28) and `Cpp17MoveAssignable` (Table 30) requirements.

Effects: Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first, last - 1)` into a heap with respect to `comp` and `proj`.

Returns: `last` for the overloads in namespace `ranges`.

Complexity: At most $2 \log(last - first)$ comparisons and twice as many projections.

25.8.8.4 make_heap

```cpp
template<class RandomAccessIterator>
constexpr void make_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
constexpr void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                        Compare comp);
```

```cpp
template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
         class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr I
ranges::make_heap(I first, S last, Comp comp = {}, Proj proj = {});
```

```cpp
template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr borrowed_iterator_t<R>
ranges::make_heap(R&& r, Comp comp = {}, Proj proj = {});
```

Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

Preconditions: For the overloads in namespace `std`, the type of `*first` meets the `Cpp17MoveConstructible` (Table 28) and `Cpp17MoveAssignable` (Table 30) requirements.

Effects: Constructs a heap with respect to `comp` and `proj` out of the range `[first, last)`.

Returns: `last` for the overloads in namespace `ranges`.

Complexity: At most $3(last - first)$ comparisons and twice as many projections.

25.8.8.5 sort_heap

```cpp
template<class RandomAccessIterator>
constexpr void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
```
template<class RandomAccessIterator, class Compare>
   constexpr void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                            Compare comp);

template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
          class Proj = identity>
   requires sortable<I, Comp, Proj>
   constexpr I
   ranges::sort_heap(I first, S last, Comp comp = {}, Proj proj = {});

template<random_access_range R, class Comp = ranges::less, class Proj = identity>
   requires sortable<iterator_t<R>, Comp, Proj>
   constexpr borrowed_iterator_t<R>
   ranges::sort_heap(R&& r, Comp comp = {}, Proj proj = {});

Let \( \text{comp} \) be \text{less\{\}} and \( \text{proj} \) be \text{identity\{\}} for the overloads with no parameters by those names.

Preconditions: The range \([\text{first}, \text{last})\) is a valid heap with respect to \( \text{comp} \) and \( \text{proj} \). For the overloads in namespace \text{std}, \text{RandomAccessIterator} meets the \text{Cpp17ValueSwappable} requirements (16.4.4.3) and the type of \*\text{first} meets the \text{Cpp17MoveConstructible} (Table 28) and \text{Cpp17MoveAssignable} (Table 30) requirements.

Effects: Sorts elements in the heap \([\text{first}, \text{last})\) with respect to \( \text{comp} \) and \( \text{proj} \).

Returns: \text{last} for the overloads in namespace \text{ranges}.

Complexity: At most \( 2N \log N \) comparisons, where \( N = \text{last} - \text{first} \), and twice as many projections.

### 25.8.8.6 is_heap

[is.heap]

template<class RandomAccessIterator>
   constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

Effects: Equivalent to: \( \text{return is_heap_until(first, last) == last;} \)

template<class ExecutionPolicy, class RandomAccessIterator>
   bool is_heap(ExecutionPolicy&& exec,
                RandomAccessIterator first, RandomAccessIterator last);

Effects: Equivalent to:
   \( \text{return is_heap_until(std::forward<ExecutionPolicy>(exec), first, last) == last;} \)

template<class RandomAccessIterator, class Compare>
   constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                         Compare comp);

Effects: Equivalent to: \( \text{return is_heap_until(first, last, comp) == last;} \)

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
   bool is_heap(ExecutionPolicy&& exec,
                RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);

Effects: Equivalent to:
   \( \text{return is_heap_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last;} \)

template<random_access_iterator I, sentinel_for<I> S, class Proj = identity,
          indirect_strict_weak_order<Projected]<I, Proj>> Comp = ranges::less>
   constexpr bool ranges::is_heap(I first, S last, Comp comp = {}, Proj proj = {});

template<random_access_range R, class Proj = identity,
          indirect_strict_weak_order<Projected<iterator_t<R>>, Proj>> Comp = ranges::less>
   constexpr bool ranges::is_heap(R&& r, Comp comp = {}, Proj proj = {});

Effects: Equivalent to: \( \text{return ranges::is_heap_until(first, last, comp, proj) == last;} \)

template<class RandomAccessIterator>
   constexpr RandomAccessIterator
   is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
    RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec,
                  RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
    constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
    RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec,
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);

template<random_access_iterator I, sentinel_for<I> S, class Proj = identity,
          indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
    constexpr I ranges::is_heap_until(I first, S last, Comp comp = {}, Proj proj = {});

template<random_access_range R, class Proj = identity,
          indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr borrowed_iterator_t<R>
    ranges::is_heap_until(R&& r, Comp comp = {}, Proj proj = {});

6 Let comp be less{} and proj be identity{} for the overloads with no parameters by those names.
7 Returns: The last iterator i in [first, last] for which the range [first, i) is a heap with respect to comp and proj.
8 Complexity: Linear.

25.8.9 Minimum and maximum [alg.min.max]

template<class T>
    constexpr const T& min(const T& a, const T& b);

template<class T, class Compare>
    constexpr const T& min(const T& a, const T& b, Compare comp);

template<class T, class Proj = identity,
          indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
    constexpr const T& ranges::min(const T& a, const T& b, Comp comp = {}, Proj proj = {});
1 Preconditions: For the first form, T meets the Cpp17LessThanComparable requirements (Table 26).
2 Returns: The smaller value. Returns the first argument when the arguments are equivalent.
3 Complexity: Exactly one comparison and two applications of the projection, if any.
4 Remarks: An invocation may explicitly specify an argument for the template parameter T of the overloads in namespace std.

template<class T>
    constexpr T min(initializer_list<T> r);

template<class T, class Compare>
    constexpr T min(initializer_list<T> r, Compare comp);

template<copyable T, class Proj = identity,
          indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
    constexpr T ranges::min(initializer_list<T> r, Comp comp = {}, Proj proj = {});
2 Preconditions: ranges::distance(r) > 0. For the overloads in namespace std, T meets the Cpp17CopyConstructible requirements. For the first form, T meets the Cpp17LessThanComparable requirements (Table 26).
Returns: The smallest value in the input range. Returns a copy of the leftmost element when several elements are equivalent to the smallest.

Complexity: Exactly `ranges::distance(r) - 1` comparisons and twice as many applications of the projection, if any.

Remarks: An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

```cpp
template<class T>
constexpr const T& max(const T& a, const T& b);
```

```cpp
template<class T, class Compare>
constexpr const T& max(const T& a, const T& b, Compare comp);
```

```cpp
template<class T, class Proj = identity,
indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
constexpr const T& ranges::max(const T& a, const T& b, Comp comp = {}, Proj proj = {});
```

Preconditions: For the first form, `T` meets the `Cpp17LessThanComparable` requirements (Table 26).

Returns: The larger value. Returns the first argument when the arguments are equivalent.

Complexity: Exactly one comparison and two applications of the projection, if any.

Remarks: An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

```cpp
template<class T>
constexpr T max(initializer_list<T> r);
```

```cpp
template<class T, class Compare>
constexpr T max(initializer_list<T> r, Compare comp);
```

```cpp
template<copyable T, class Proj = identity,
indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>
requires indirectly_copyable_storable<iterator_t<R>, range_value_t<R>*>
constexpr range_value_t<R>
ranges::max(R&& r, Comp comp = {}, Proj proj = {});
```

Preconditions: `ranges::distance(r) > 0`. For the overloads in namespace `std`, `T` meets the `Cpp17CopyConstructible` requirements. For the first form, `T` meets the `Cpp17LessThanComparable` requirements (Table 26).

Returns: The largest value in the input range. Returns a copy of the leftmost element when several elements are equivalent to the largest.

Complexity: Exactly `ranges::distance(r) - 1` comparisons and twice as many applications of the projection, if any.

Remarks: An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

```cpp
template<class T>
constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
```

```cpp
template<class T, class Compare>
constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

```cpp
template<class T, class Proj = identity,
indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>
ranges::minmax_result<const T&>
ranges::minmax(const T& a, const T& b, Comp comp = {}, Proj proj = {});
```

Preconditions: For the first form, `T` meets the `Cpp17LessThanComparable` requirements (Table 26).

Returns: `{b, a}` if `b` is smaller than `a`, and `{a, b}` otherwise.

Complexity: Exactly one comparison and two applications of the projection, if any.
Remarks: An invocation may explicitly specify an argument for the template parameter T of the overloads in namespace std.

```cpp
template<class T>
constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);
```

```cpp
template<copyable T, class Proj = identity,
  indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
ranges::minmax_result<T>
ranges::minmax(initializer_list<T> r, Compare comp = {}, Proj proj = {});
```

```cpp
template<input_range R, class Proj = identity,
  indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>
requires indirectly_copyable_storable<iterator_t<R>, range_value_t<R>>
ranges::minmax_result<range_value_t<R>>
ranges::minmax(R&& r, Compare comp = {}, Proj proj = {});
```

Preconditions: ranges::distance(r) > 0. For the overloads in namespace std, T meets the Cpp17-CopyConstructible requirements. For the first form, type T meets the Cpp17LessThanComparable requirements (Table 26).

Returns: Let X be the return type. Returns X{x, y}, where x is a copy of the leftmost element with the smallest value and y a copy of the rightmost element with the largest value in the input range.

Complexity: At most \((3/2)\)ranges::distance(r) applications of the corresponding predicate and twice as many applications of the projection, if any.

Remarks: An invocation may explicitly specify an argument for the template parameter T of the overloads in namespace std.

```cpp
template<class ForwardIterator>
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator min_element(ExecutionPolicy&& exec,
  ForwardIterator first, ForwardIterator last);
```

```cpp
template<class ForwardIterator, class Compare>
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
  Compare comp);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator min_element(ExecutionPolicy&& exec,
  ForwardIterator first, ForwardIterator last, Compare comp);
```

```cpp
template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
  indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
constexpr I ranges::min_element(I first, S last, Compare comp = {}, Proj proj = {});
```

```cpp
template<forward_range R, class Proj = identity,
  indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>
ranges::min_element(borrowed_iterator_t<R>
ranges::min_element(R&& r, Compare comp = {}, Proj proj = {});
```

Let comp be less{} and proj be identity{} for the overloads with no parameters by those names.

Returns: The first iterator i in the range [first, last) such that for every iterator j in the range [first, last),

\[ \text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, *j), \text{invoke}(\text{proj}, *i))) \]

is false. Returns last if first == last.

Complexity: Exactly max(last - first - 1,0) comparisons and twice as many projections.
template<class ForwardIterator, class Compare>
constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last, Compare comp);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator max_element(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, Compare comp);

template<forward_iterator I, sentinel_for<I> S, class Proj = identity, indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
constexpr I ranges::max_element(I first, S last, Comp comp = {}, Proj proj = {});

template<forward_range R, class Proj = identity, indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>
constexpr borrowed_iterator_t<R> ranges::max_element(R&& r, Comp comp = {}, Proj proj = {});

Let \( \text{comp} \) be \text{less{}} and \( \text{proj} \) be \text{identity{}} for the overloads with no parameters by those names.

Returns: The first iterator \( i \) in the range \([\text{first}, \text{last})\) such that for every iterator \( j \) in the range \([\text{first}, \text{last})\)
\[
\text{bool}(\text{invoke} (\text{comp}, \text{invoke} (\text{proj}, *i), \text{invoke} (\text{proj}, *j)))
\]
is false. Returns \( \text{last} \) if \( \text{first} == \text{last} \).

Complexity: Exactly \( \max (\text{last} - \text{first} - 1, 0) \) comparisons and twice as many projections.

template<class ForwardIterator>
constexpr pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
pair<ForwardIterator, ForwardIterator> minmax_element(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
constexpr pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator> minmax_element(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, Compare comp);

template<forward_iterator I, sentinel_for<I> S, class Proj = identity, indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
constexpr ranges::minmax_result<I> ranges::minmax_element(I first, S last, Comp comp = {}, Proj proj = {});

template<forward_range R, class Proj = identity, indirect_strict_weak_order<projected<iterator_t<R>, Proj>> Comp = ranges::less>
constexpr ranges::minmax_result<borrowed_iterator_t<R>> ranges::minmax_element(R&& r, Comp comp = {}, Proj proj = {});

Returns: \( \{\text{first}, \text{first}\} \) if \([\text{first}, \text{last})\) is empty, otherwise \( \{m, M\} \), where \( m \) is the first iterator in \([\text{first}, \text{last})\) such that no iterator in the range refers to a smaller element, and where \( M \) is the last iterator in \([\text{first}, \text{last})\) such that no iterator in the range refers to a larger element.

Complexity: Let \( N \) be \( \text{last} - \text{first} \). At most \( \max (\frac{3}{2} N, 0) \) comparisons and twice as many applications of the projection, if any.

25.8.10 Bounded value [alg.clamp]

template<class T>
constexpr const T& clamp(const T& v, const T& lo, const T& hi);

template<class T, class Compare>
constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp);

This behavior intentionally differs from \text{max_element{}}.
template<class T, class Proj = identity,
   indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
constexpr const T&
ranges::clamp(const T& v, const T& lo, const T& hi, Comp comp = {}, Proj proj = {});

1 Let comp be less{} for the overloads with no parameter comp, and let proj be identity{} for the
   overloads with no parameter proj.
2 Preconditions: bool(invoke(comp, invoke(proj, hi), invoke(proj, lo))) is false. For the first
   form, type T meets the Cpp17LessThanComparable requirements (Table 26).
3 Returns: lo if bool(invoke(comp, invoke(proj, v), invoke(proj, lo))) is true, hi if bool(
   invoke(comp, invoke(proj, hi), invoke(proj, v))) is true, otherwise v.
4 [Note 1: If NaN is avoided, T can be a floating-point type. — end note]
5
Complexity: At most two comparisons and three applications of the projection.

25.8.11 Lexicographical comparison [alg.lex.comparison]

template<class InputIterator1, class InputIterator2>
constexpr bool
lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
bool
lexicographical_compare(ExecutionPolicy&& exec,
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
constexpr bool
lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2,
   Compare comp);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
   class Compare>
bool
lexicographical_compare(ExecutionPolicy&& exec,
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2,
   Compare comp);

template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
   class Proj1 = identity, class Proj2 = identity,
   indirect_strict_weak_order<projected<I1, Proj1>,
   projected<I2, Proj2>> Comp = ranges::less>
constexpr bool
ranges::lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
   Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, input_range R2, class Proj1 = identity,
   class Proj2 = identity,
   indirect_strict_weak_order<projected<iterator_t<R1>, Proj1>,
   projected<iterator_t<R2>, Proj2>> Comp = ranges::less>
constexpr bool
ranges::lexicographical_compare(R1&& r1, R2&& r2, Comp comp = {},
   Proj1 proj1 = {}, Proj2 proj2 = {});

1 Returns: true if and only if the sequence of elements defined by the range [first1, last1) is
   lexicographically less than the sequence of elements defined by the range (first2, last2).
2 Complexity: At most 2 min(last1 - first1, last2 - first2) applications of the corresponding
   comparison and each projection, if any.
3 Remarks: If two sequences have the same number of elements and their corresponding elements (if
   any) are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a
   proper prefix of the other, then the shorter sequence is lexicographically less than the longer sequence.
Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

Example 1: ranges::lexicographical_compare(I1, S1, I2, S2, Comp, Proj1, Proj2) could be implemented as:

```cpp
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
    if (invoke(comp, invoke(proj1, *first1), invoke(proj2, *first2))) return true;
    if (invoke(comp, invoke(proj2, *first2), invoke(proj1, *first1))) return false;
} return first1 == last1 & first2 != last2;
```
—end example

Note 1: An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence. — end note

### 25.8.12 Three-way comparison algorithms [alg.three.way]

```cpp
template<class InputIterator1, class InputIterator2, class Cmp>
constexpr auto
lexicographical_compare_three_way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2,
                                 Cmp comp)
-> decltype(comp(*b1, *b2));
```

1. **Mandates:** decltype(comp(*b1, *b2)) is a comparison category type.

2. **Effects:** Lexicographically compares two ranges and produces a result of the strongest applicable comparison category type. Equivalent to:

```cpp
for ( ; b1 != e1 && b2 != e2; void(++b1), void(++b2) )
    if (auto cmp = comp(*b1,*b2); cmp != 0)
        return cmp;
    return b1 != e1 ? strong_ordering::greater :
        b2 != e2 ? strong_ordering::less :
            strong_ordering::equal;
```

```cpp
template<class InputIterator1, class InputIterator2>
constexpr auto
lexicographical_compare_three_way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2);
```

3. **Effects:** Equivalent to:

```cpp
return lexicographical_compare_three_way(b1, e1, b2, e2, compare_three_way());
```

### 25.8.13 Permutation generators [alg.permutation.generators]

```cpp
template<class BidirectionalIterator>
constexpr bool next_permutation(BidirectionalIterator first,
                                 BidirectionalIterator last);
```

```cpp
template<class BidirectionalIterator, class Compare>
constexpr bool next_permutation(BidirectionalIterator first,
                                 BidirectionalIterator last, Compare comp);
```

```cpp
template<bidirectional_iterator I, sentinel_for<I> S, class Comp = ranges::less,
         class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr ranges::next_permutation_result<I>
ranges::next_permutation(I first, S last, Comp comp = {}, Proj proj = {});
```

```cpp
template<bidirectional_iterator_t<R>, class Comp = ranges::less,
         class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr ranges::next_permutation_result<borrowed_iterator_t<R>>
ranges::next_permutation(R&& r, Comp comp = {}, Proj proj = {});
```

Let comp be less{} and proj be identity{} for overloads with no parameters by those names.
Preconditions: For the overloads in namespace std, BidirectionalIterator meets the Cpp17ValueSwappable requirements (16.4.4.3).

Effects: Takes a sequence defined by the range [first, last) and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to comp and proj. If no such permutation exists, transforms the sequence into the first permutation; that is, the ascendingly-sorted one.

Returns: Let B be true if a next permutation was found and otherwise false. Returns:

(4.1) — B for the overloads in namespace std.
(4.2) — { last, B } for the overloads in namespace ranges.

Complexity: At most (last - first) / 2 swaps.

```cpp
template<class BidirectionalIterator>
constexpr bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);
```

```cpp
template<class BidirectionalIterator, class Compare>
constexpr bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);
```

```cpp
template<bidirectional_iterator I, sentinel_for<I> S, class Comp = ranges::less, class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr ranges::prev_permutation_result<I> ranges::prev_permutation(I first, S last, Comp comp = {}, Proj proj = {});
```

```cpp
template<class BidirectionalRange R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr ranges::prev_permutation_result<borrowed_iterator_t<R>> ranges::prev_permutation(R&& r, Comp comp = {}, Proj proj = {});
```

Let comp be less{} and proj be identity{} for overloads with no parameters by those names.

Preconditions: For the overloads in namespace std, BidirectionalIterator meets the Cpp17ValueSwappable requirements (16.4.4.3).

Effects: Takes a sequence defined by the range [first, last) and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to comp and proj. If no such permutation exists, transforms the sequence into the last permutation; that is, the descendingly-sorted one.

Returns: Let B be true if a previous permutation was found and otherwise false. Returns:

(9.1) — B for the overloads in namespace std.
(9.2) — { last, B } for the overloads in namespace ranges.

Complexity: At most (last - first) / 2 swaps.

25.9 Header <numeric> synopsis

```cpp
namespace std {

// 25.10.3, accumulate
template<class InputIterator, class T>
constexpr T accumulate(InputIterator first, InputIterator last, T init);
template<class InputIterator, class T, class BinaryOperation>
constexpr T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation binary_op);

// 25.10.4, reduce
template<class InputIterator>
constexpr typename iterator_traits<InputIterator>::value_type reduce(InputIterator first, InputIterator last);
template<class InputIterator, class T>
constexpr T reduce(InputIterator first, InputIterator last, T init);
```

§ 25.9
template<class InputIterator, class T, class BinaryOperation>
constexpr T reduce(InputIterator first, InputIterator last, T init,
   BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator>
typename iterator_traits<ForwardIterator>::value_type
reduce(ExecutionPolicy& exec,
   // see 25.3.5
   ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class T>
T reduce(ExecutionPolicy& exec,
   // see 25.3.5
   ForwardIterator first, ForwardIterator last, T init);

template<class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
T reduce(ExecutionPolicy& exec,
   // see 25.3.5
   ForwardIterator first, ForwardIterator last, T init, BinaryOperation binary_op);

// 25.10.5, inner product

template<class InputIterator1, class InputIterator2, class T>
constexpr T inner_product(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, T init);

template<class InputIterator1, class InputIterator2, class T,
   class BinaryOperation1, class BinaryOperation2>
constexpr T inner_product(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, T init,
   BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);

// 25.10.6, transform reduce

template<class InputIterator1, class InputIterator2, class T>
constexpr T transform_reduce(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, T init);

template<class InputIterator1, class InputIterator2, class T,
   class BinaryOperation1, class BinaryOperation2>
constexpr T transform_reduce(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, T init,
   BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);

template<class InputIterator, class T,
   class BinaryOperation, class UnaryOperation>
constexpr T transform_reduce(InputIterator first, InputIterator last, T init,
   BinaryOperation binary_op, UnaryOperation unary_op);

template<class ExecutionPolicy,
   class ForwardIterator1, class ForwardIterator2, class T>
T transform_reduce(ExecutionPolicy& exec,
   // see 25.3.5
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, T init);

template<class ExecutionPolicy,
   class ForwardIterator1, class ForwardIterator2, class T,
   class BinaryOperation1, class BinaryOperation2>
T transform_reduce(ExecutionPolicy& exec,
   // see 25.3.5
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, T init,
   BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);

template<class ExecutionPolicy,
   class ForwardIterator, class T,
   class BinaryOperation, class UnaryOperation>
T transform_reduce(ExecutionPolicy& exec,
   // see 25.3.5
   ForwardIterator first, ForwardIterator last, T init,
   BinaryOperation binary_op, UnaryOperation unary_op);

// 25.10.7, partial sum

template<class InputIterator, class OutputIterator>
constexpr OutputIterator
partial_sum(InputIterator first, InputIterator last,
   OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryOperation>
constexpr OutputIterator
partial_sum(InputIterator first, InputIterator last,
   OutputIterator result, BinaryOperation binary_op);
// 25.10.8, exclusive scan

template<class InputIterator, class OutputIterator, class T>
constexpr OutputIterator
exclusive_scan(InputIterator first, InputIterator last,
OutputIterator result, T init);

template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
constexpr OutputIterator
exclusive_scan(InputIterator first, InputIterator last,
OutputIterator result, T init, BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
ForwardIterator2
exclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, T init);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation>
ForwardIterator2
exclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, T init, BinaryOperation binary_op);

// 25.10.9, inclusive scan

template<class InputIterator, class OutputIterator>
constexpr OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryOperation>
constexpr OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
OutputIterator result, BinaryOperation binary_op);

template<class InputIterator, class OutputIterator, class BinaryOperation, class T>
constexpr OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
OutputIterator result, BinaryOperation binary_op, T init);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
inclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation>
ForwardIterator2
inclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation, class UnaryOperation>
ForwardIterator2
inclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryOperation binary_op, T init);

// 25.10.10, transform exclusive scan

template<class InputIterator, class OutputIterator, class T,
class BinaryOperation, class UnaryOperation>
constexpr OutputIterator
transform_exclusive_scan(InputIterator first, InputIterator last,
OutputIterator result, T init,
BinaryOperation binary_op, UnaryOperation unary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T,
class BinaryOperation, class UnaryOperation>
ForwardIterator2
transform_exclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, T init,
BinaryOperation binary_op, UnaryOperation unary_op);

// 25.10.11, transform inclusive scan
template<class InputIterator, class OutputIterator,
class BinaryOperation, class UnaryOperation>
constexpr OutputIterator
transform_inclusive_scan(InputIterator first, InputIterator last,
OutputIterator result,
BinaryOperation binary_op, UnaryOperation unary_op);

template<class InputIterator, class OutputIterator,
class BinaryOperation, class UnaryOperation, class T>
constexpr OutputIterator
transform_inclusive_scan(InputIterator first, InputIterator last,
OutputIterator result,
BinaryOperation binary_op, UnaryOperation unary_op, T init);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryOperation, class UnaryOperation>
ForwardIterator2
transform_inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryOperation binary_op,
UnaryOperation unary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryOperation, class UnaryOperation, class T>
ForwardIterator2
transform_inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryOperation binary_op,
UnaryOperation unary_op, T init);

// 25.10.12, adjacent difference
template<class InputIterator, class OutputIterator>
constexpr OutputIterator
adjacent_difference(InputIterator first, InputIterator last,
OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryOperation>
constexpr OutputIterator
adjacent_difference(InputIterator first, InputIterator last,
OutputIterator result, BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryOperation>
ForwardIterator2
adjacent_difference(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryOperation>
ForwardIterator2
adjacent_difference(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryOperation binary_op);

// 25.10.13, iota
template<class ForwardIterator, class T>
constexpr void iota(ForwardIterator first, ForwardIterator last, T value);

// 25.10.14, greatest common divisor
template<class M, class N>
constexpr common_type_t<M,N> gcd(M m, N n);

// 25.10.15, least common multiple
template<class M, class N>
constexpr common_type_t<M,N> lcm(M m, N n);
// 25.10.16, midpoint
template<class T>
constexpr T midpoint(T a, T b) noexcept;
constexpr T* midpoint(T* a, T* b);

25.10 Generalized numeric operations

25.10.1 General

[Note 1: The use of closed ranges as well as semi-open ranges to specify requirements throughout 25.10 is intentional. —end note]

25.10.2 Definitions

1 Define GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN) as follows:
   \[(1.1)\] a1 when N is 1, otherwise
   \[(1.2)\] op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK),
            GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN)) for any K where 1 < K + 1 = M ≤ N.

2 Define GENERALIZED_SUM(op, a1, ..., aN) as GENERALIZED_NONCOMMUTATIVE_SUM(op, b1, ..., bN),
   where b1, ..., bN may be any permutation of a1, ..., aN.

25.10.3 Accumulate

template<class InputIterator, class T>
constexpr T accumulate(InputIterator first, InputIterator last, T init);

Effects: Computes its result by initializing the accumulator acc with the initial value init and then modifies it with
acc = std::move(acc) + *i or acc = binary_op(std::move(acc), *i) for every iterator i in the range [first, last) in order.

25.10.4 Reduce

template<class InputIterator, class T>
constexpr T reduce(InputIterator first, InputIterator last, T init);

Effects: Equivalent to:
return reduce(first, last, init);

241) The use of fully closed ranges is intentional.
242) accumulate is similar to the APL reduction operator and Common Lisp reduce function, but it avoids the difficulty of
defining the result of reduction on an empty sequence by always requiring an initial value.

§ 25.10.4 1145
```cpp
template<class ExecutionPolicy, class ForwardIterator, class T>
T reduce(ExecutionPolicy&& exec,
         ForwardIterator first, ForwardIterator last, T init);
```

**Effects:** Equivalent to:

```cpp
return reduce(std::forward<ExecutionPolicy>(exec), first, last, init, plus<>());
```

```cpp
template<class InputIterator, class T, class BinaryOperation>
constexpr T reduce(InputIterator first, InputIterator last, T init,
                   BinaryOperation binary_op);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
T reduce(ExecutionPolicy&& exec,
         ForwardIterator first, ForwardIterator last, T init,
         BinaryOperation binary_op);
```

**Mandates:** All of

- binary_op(init, *first),
- binary_op(*first, init),
- binary_op(init, init), and
- binary_op(*first, *first)

are convertible to T.

**Preconditions:**

1. T meets the Cpp17MoveConstructible (Table 28) requirements.
2. binary_op neither invalidates iterators or subranges, nor modifies elements in the range [first, last].

**Returns:** GENERALIZED_SUM(binary_op, init, *i, ...) for every i in [first, last).

**Complexity:** $O(last - first)$ applications of binary_op.

**Note 1:** The difference between reduce and accumulate is that reduce applies binary_op in an unspecified order, which yields a nondeterministic result for non-associative or non-commutative binary_op such as floating-point addition. — end note

### 25.10.5 Inner product

```cpp
template<class InputIterator1, class InputIterator2, class T>
constexpr T inner_product(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, T init);
```

```cpp
template<class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
constexpr T inner_product(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, T init,
                           BinaryOperation1 binary_op1,
                           BinaryOperation2 binary_op2);
```

**Preconditions:** T meets the Cpp17CopyConstructible (Table 29) and Cpp17CopyAssignable (Table 31) requirements. In the ranges [first1, last1] and [first2, first2 + (last1 - first1)] binary_op1 and binary_op2 neither modifies elements nor invalidates iterators or subranges.

**Effects:** Computes its result by initializing the accumulator acc with the initial value init and then modifying it with acc = std::move(acc) + (*i1) * (*i2) or acc = binary_op1(std::move(acc), binary_op2(*i1, *i2)) for every iterator i1 in the range [first1, last1) and iterator i2 in the range [first2, first2 + (last1 - first1)) in order.

### 25.10.6 Transform reduce

```cpp
template<class InputIterator1, class InputIterator2, class T>
constexpr T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2,
                             T init);
```

**Effects:** Equivalent to:

243) The use of fully closed ranges is intentional.
return transform_reduce(first1, last1, first2, init, plus<>(), multiplies<>());

template<class ExecutionPolicy, 
    class ForwardIterator1, class ForwardIterator2, class T>
T transform_reduce(ExecutionPolicy&& exec, 
    ForwardIterator1 first1, ForwardIterator1 last1, 
    ForwardIterator2 first2, 
    T init);

Effects: Equivalent to:
return transform_reduce(std::forward<ExecutionPolicy>(exec), 
    first1, last1, first2, init, plus<>(), multiplies<>());

template<class InputIterator1, class InputIterator2, class T, 
    class BinaryOperation1, class BinaryOperation2>
constexpr T transform_reduce(InputIterator1 first1, InputIterator1 last1, 
    InputIterator2 first2, 
    T init, 
    BinaryOperation1 binary_op1, 
    BinaryOperation2 binary_op2);

template<class ExecutionPolicy, 
    class ForwardIterator1, class ForwardIterator2, class T, 
    class BinaryOperation1, class BinaryOperation2>
T transform_reduce(ExecutionPolicy&& exec, 
    ForwardIterator1 first1, ForwardIterator1 last1, 
    ForwardIterator2 first2, 
    T init, 
    BinaryOperation1 binary_op1, 
    BinaryOperation2 binary_op2);

Mandates: All of
— binary_op1(init, init),
— binary_op1(init, binary_op2(*first1, *first2)),
— binary_op1(binary_op2(*first1, *first2), init), and
— binary_op1(binary_op2(*first1, *first2), binary_op2(*first1, *first2))
are convertible to T.

Preconditions:
— T meets the Cpp17MoveConstructible (Table 28) requirements.
— Neither binary_op1 nor binary_op2 invalidates subranges, nor modifies elements in the ranges [first1, last1] and [first2, first2 + (last1 - first1)].

Returns:
GENERALIZED_SUM(binary_op1, init, binary_op2(*i, *(first2 + (i - first1))), ...)
for every iterator i in [first1, last1).

Complexity: \(O(last1 - first1)\) applications each of binary_op1 and binary_op2.

template<class InputIterator, class T, 
    class BinaryOperation, class UnaryOperation>
constexpr T transform_reduce(InputIterator first, InputIterator last, T init, 
    BinaryOperation binary_op, UnaryOperation unary_op);

template<class ExecutionPolicy, 
    class ForwardIterator, class T, 
    class BinaryOperation, class UnaryOperation>
T transform_reduce(ExecutionPolicy&& exec, 
    ForwardIterator first, ForwardIterator last, 
    T init, BinaryOperation binary_op, UnaryOperation unary_op);

Mandates: All of
— binary_op(init, init),
— binary_op(init, unary_op(*first)),

§ 25.10.6
(7.3) — binary_op(unary_op(*first), init), and
(7.4) — binary_op(unary_op(*first), unary_op(*first))
are convertible to T.

Preconditions:
(8.1) — T meets the Cpp17MoveConstructible (Table 28) requirements.
(8.2) — Neither unary_op nor binary_op invalidates subranges, nor modifies elements in the range [first, last].

Returns:
GENERALIZED_SUM(binary_op, init, unary_op(*i), ...)
for every iterator i in [first, last).

Complexity: $\Theta(la - fir) \times 1$ applications each of unary_op and binary_op.

[Note 1: transform_reduce does not apply unary_op to init. — end note]

25.10.7 Partial sum [partial.sum]

partial_sum(InputIterator first, InputIterator last, OutputIterator result);

partial_sum(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation binary_op);

Mandates: InputIterator’s value type is constructible from *first. The result of the expression std::move(acc) + *i or binary_op(std::move(acc), *i) is implicitly convertible to InputIterator’s value type. acc is writable (23.3.1) to result.

Preconditions: In the ranges [first, last] and [result, result + (last - first)] binary_op neither modifies elements nor invalidates iterators or subranges.

Effects: For a non-empty range, the function creates an accumulator acc whose type is InputIterator’s value type, initializes it with *first, and assigns the result to *result. For every iterator i in [first + 1, last) in order, acc is then modified by acc = std::move(acc) + *i or acc = binary_op(std::move(acc), *i) and the result is assigned to *(result + (i - first)).

Returns: result + (last - first).

Complexity: Exactly (last - first) - 1 applications of the binary operation.

Remarks: result may be equal to first.

25.10.8 Exclusive scan [exclusive.scan]

exclusive_scan(InputIterator first, InputIterator last, OutputIterator result, T init);

effects: Equivalent to:
return exclusive_scan(first, last, result, init, plus<>());

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, T init);

effects: Equivalent to:
return exclusive_scan(std::forward<ExecutionPolicy>(exec),
first, last, result, init, plus<>());

244) The use of fully closed ranges is intentional.
template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
constexpr OutputIterator
exclusive_scan(InputIterator first, InputIterator last, 
OutputIterator result, T init, BinaryOperation binary_op);

template<class ExecutionPolicy, 
class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation>
ForwardIterator2
exclusive_scan(ExecutionPolicy&& exec, 
ForwardIterator1 first, ForwardIterator1 last, 
ForwardIterator2 result, T init, BinaryOperation binary_op);

Mandates: All of
— binary_op(init, init),
— binary_op(init, *first), and
— binary_op(*first, *first)
are convertible to T.

Preconditions:
— T meets the Cpp17MoveConstructible (Table 28) requirements.
— binary_op neither invalidates iterators or subranges, nor modifies elements in the ranges [first, last] or [result, result + (last - first)].

Effects: For each integer K in [0, last - first) assigns through result + K the value of:

\[
\text{GENERALIZED NONCOMMUTATIVE SUM} \\
binary_op, \init, *(\text{first} + 0), *(\text{first} + 1), \ldots, *(\text{first} + K - 1))
\]

Returns: The end of the resulting range beginning at result.

Complexity: \(\Theta(l_{ast} - first)\) applications of binary_op.

Remarks: result may be equal to first.

[Note 1: The difference between exclusive_scan and inclusive_scan is that exclusive_scan excludes the \(i^{th}\) input element from the \(i^{th}\) sum. If binary_op is not mathematically associative, the behavior of exclusive_scan might be nondeterministic. — end note]
template<class InputIterator, class OutputIterator, class BinaryOperation, class T>
constexpr OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
OutputIterator result, BinaryOperation binary_op, T init);

template<class ExecutionPolicy,
class ForwardIterator1, class ForwardIterator2, class BinaryOperation, class T>
ForwardIterator2
inclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryOperation binary_op, T init);

Let \( U \) be the value type of \texttt{decltype(first)}.

\textbf{Mandates:} If \texttt{init} is provided, all of
\begin{enumerate}
\item binary\_op(init, init),
\item binary\_op(init, *first), and
\item binary\_op(*first, *first)
\end{enumerate}
are convertible to \( T \); otherwise, \texttt{binary\_op(*first, *first)} is convertible to \( U \).

\textbf{Preconditions:}
\begin{enumerate}
\item If \texttt{init} is provided, \( T \) meets the \texttt{Cpp17MoveConstructible} (Table 28) requirements; otherwise, \( U \) meets the \texttt{Cpp17MoveConstructible} requirements.
\item \texttt{binary\_op} neither invalidates iterators or subranges, nor modifies elements in the ranges \([\texttt{first}, \texttt{last})\) or \([\texttt{result}, \texttt{result} + (\texttt{last} - \texttt{first})] \).
\end{enumerate}

\textbf{Effects:} For each integer \( K \) in \([0, \texttt{last} - \texttt{first})\) assigns through \texttt{result} + \( K \) the value of
\begin{enumerate}
\item \texttt{GENERALIZED\_NONCOMMUTATIVE\_SUM}(
\texttt{binary\_op, init, *(first + 0), *(first + 1), \ldots, *(first + K)}
if \texttt{init} is provided, or
\item \texttt{GENERALIZED\_NONCOMMUTATIVE\_SUM}(
\texttt{binary\_op, *(first + 0), *(first + 1), \ldots, *(first + K)}
otherwise.
\end{enumerate}

\textbf{Returns:} The end of the resulting range beginning at \texttt{result}.

\textbf{Complexity:} \( \Theta(\texttt{last} - \texttt{first}) \) applications of \texttt{binary\_op}.

\textbf{Remarks:} \texttt{result} may be equal to \texttt{first}.

\textbf{Note 1:} The difference between \texttt{exclusive\_scan} and \texttt{inclusive\_scan} is that \texttt{inclusive\_scan} includes the \( i \)th input element in the \( i \)th sum. If \texttt{binary\_op} is not mathematically associative, the behavior of \texttt{inclusive\_scan} might be nondeterministic. — end note

\section{25.10.10 Transform exclusive scan}

\texttt{template<class InputIterator, class OutputIterator, class T, class BinaryOperation, class UnaryOperation>}
\texttt{constexpr OutputIterator}
\texttt{transform_exclusive_scan(InputIterator first, InputIterator last,}
\texttt{OutputIterator result, T init,}
\texttt{BinaryOperation binary\_op, UnaryOperation unary\_op);}
binary_op(unary_op(*first), unary_op(*first))

are convertible to T.

Preconditions:

1. T meets the Cpp17MoveConstructible (Table 28) requirements.
2. Neither unary_op nor binary_op invalidates iterators or subranges, nor modifies elements in the ranges [first, last) or [result, result + (last - first)].

Effects: For each integer K in [0, last - first) assigns through result + K the value of:

\[ \text{GENERALIZED_NONCOMMUTATIVE_SUM} \]

\[
\text{binary_op}, \text{init}, \newline
\text{unary_op}(*(\text{first} + 0)), \text{unary_op}(*(\text{first} + 1)), \ldots, \text{unary_op}(*(\text{first} + K - 1))
\]

Returns: The end of the resulting range beginning at result.

Complexity: \( O(last - first) \) applications each of unary_op and binary_op.

Remarks: result may be equal to first.

[Note 1: The difference between \text{transform_exclusive_scan} and \text{transform_inclusive_scan} is that transform_exclusive_scan excludes the \( i^{th} \) input element from the \( i^{th} \) sum. If binary_op is not mathematically associative, the behavior of transform_exclusive_scan might be nondeterministic. transform_exclusive_scan does not apply unary_op to init. — end note]
Preconditions:

— If `init` is provided, `T` meets the `Cpp17MoveConstructible` (Table 28) requirements; otherwise, `U` meets the `Cpp17MoveConstructible` requirements.

— Neither `unary_op` nor `binary_op` invalidates iterators or subranges, nor modifies elements in the ranges `[first, last]` or `[result, result + (last - first)]`.

Effects: For each integer `K` in `[0, last - first)` assigns through `result + K` the value of

— `GENERALIZED_NONCOMMUTATIVE_SUM`

  `binary_op, init, unary_op(*(first + 0)), unary_op(*(first + 1)), ..., unary_op(*(first + K))` if `init` is provided, or

— `GENERALIZED_NONCOMMUTATIVE_SUM`

  `binary_op, unary_op(*(first + 0)), unary_op(*(first + 1)), ..., unary_op(*(first + K))` otherwise.

Returns: The end of the resulting range beginning at `result`.

Complexity: \( \Theta(\text{last - first}) \) applications each of `unary_op` and `binary_op`.

Remarks: `result` may be equal to `first`.

[Note 1: The difference between `transform_exclusive_scan` and `transform_inclusive_scan` is that `transform_inclusive_scan` includes the \( i \)th input element in the \( i \)th sum. If `binary_op` is not mathematically associative, the behavior of `transform_inclusive_scan` might be nondeterministic. `transform_inclusive_scan` does not apply `unary_op` to `init`. — end note]

### 25.10.12 Adjacent difference

[adjacent.difference]

```cpp
template<class InputIterator, class OutputIterator>
constexpr OutputIterator
adjacent_difference(InputIterator first, InputIterator last, 
OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
adjacent_difference(ExecutionPolicy&& exec, 
ForwardIterator1 first, ForwardIterator1 last, ForwardIterator2 result);

template<class InputIterator, class OutputIterator, class BinaryOperation>
constexpr OutputIterator
adjacent_difference(InputIterator first, InputIterator last, 
OutputIterator result, BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, 
class BinaryOperation>
ForwardIterator2
adjacent_difference(ExecutionPolicy&& exec, 
ForwardIterator1 first, ForwardIterator1 last, 
ForwardIterator2 result, BinaryOperation binary_op);
```

Let `T` be the value type of `decltype(first)`. For the overloads that do not take an argument `binary_op`, let `binary_op` be an lvalue that denotes an object of type `minus<>`.

Mandates:

— For the overloads with no `ExecutionPolicy`, `T` is constructible from `*first. acc` (defined below) is writable (23.3.1) to the `result` output iterator. The result of the expression `binary_op(val, std::move(acc))` is writable to `result`.

— For the overloads with an `ExecutionPolicy`, the result of the expressions `binary_op(*first, *first)` and `*first` are writable to `result`.

Preconditions:
For the overloads with no `ExecutionPolicy`, T meets the `Cpp17MoveAssignable` (Table 30) requirements.

For all overloads, in the ranges `[first, last]` and `[result, result + (last - first)]`, `binary_op` neither modifies elements nor invalidate iterators or subranges.

Effects: For the overloads with no `ExecutionPolicy` and a non-empty range, the function creates an accumulator `acc` of type T, initializes it with `*first`, and assigns the result to `*result`. For every iterator i in `[first + 1, last)` in order, creates an object `val` whose type is T, initializes it with `*i`, computes `binary_op(val, std::move(acc))`, assigns the result to `*(result + (i - first))`, and move assigns from `val` to `acc`.

For the overloads with an `ExecutionPolicy` and a non-empty range, performs `*result = *first`. Then, for every d in `[1, last - first - 1]`, performs `*(result + d) = binary_op(*(first + d), *(first + (d - 1)))`.

Returns: `result + (last - first)`.

Complexity: Exactly `last - first - 1` applications of the binary operation.

Remarks: For the overloads with no `ExecutionPolicy`, `result` may be equal to `first`. For the overloads with an `ExecutionPolicy`, the ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

25.10.14 Greatest common divisor [numeric.ops.gcd]

```cpp
template<class M, class N> constexpr common_type_t<M, N> gcd(M m, N n);
```

Mandates: `M` and `N` both are integer types other than `cv bool`.

Preconditions: `|m|` and `|n|` are representable as a value of `common_type_t<M, N>`.

[Note 1: These requirements ensure, for example, that `gcd(m, m) = |m|` is representable as a value of type `M`. —end note]

Returns: Zero when `m` and `n` are both zero. Otherwise, returns the greatest common divisor of `|m|` and `|n|`.

Throws: Nothing.

25.10.15 Least common multiple [numeric.ops.lcm]

```cpp
template<class M, class N> constexpr common_type_t<M, N> lcm(M m, N n);
```

Mandates: `M` and `N` both are integer types other than `cv bool`.

Preconditions: `|m|` and `|n|` are representable as a value of `common_type_t<M, N>`. The least common multiple of `|m|` and `|n|` is representable as a value of type `common_type_t<M, N>`.

Returns: Zero when either `m` or `n` is zero. Otherwise, returns the least common multiple of `|m|` and `|n|`.

Throws: Nothing.

245) The use of fully closed ranges is intentional.
25.10.16 Midpoint

template<class T>
constexpr T midpoint(T a, T b) noexcept;

Constraints: T is an arithmetic type other than bool.

Returns: Half the sum of a and b. If T is an integer type and the sum is odd, the result is rounded towards a.

Remarks: No overflow occurs. If T is a floating-point type, at most one inexact operation occurs.

template<class T>
constexpr T* midpoint(T* a, T* b);

Constraints: T is an object type.

Mandates: T is a complete type.

Preconditions: a and b point to, respectively, elements i and j of the same array object x.

[Note 1: As specified in 6.8.3, an object that is not an array element is considered to belong to a single-element array for this purpose and a pointer past the last element of an array of n elements is considered to be equivalent to a pointer to a hypothetical array element n for this purpose. — end note]

Returns: A pointer to array element i + \frac{j-i}{2} of x, where the result of the division is truncated towards zero.

25.11 Specialized <memory> algorithms

25.11.1 General

The contents specified in 25.11 are declared in the header <memory> (20.10.2).

Unless otherwise specified, if an exception is thrown in the following algorithms, objects constructed by a placement new-expression (7.6.2.8) are destroyed in an unspecified order before allowing the exception to propagate.

[Note 1: When invoked on ranges of potentially-overlapping subobjects (6.7.2), the algorithms specified in 25.11 result in undefined behavior. — end note]

Some algorithms specified in 25.11 make use of the exposition-only function voidify:

```cpp
template<class T>
constexpr void* voidify(T& obj) noexcept {
    return const_cast<void*>(static_cast<const volatile void*>(addressof(obj)));
}
```

25.11.2 Special memory concepts

Some algorithms in this subclause are constrained with the following exposition-only concepts:

```cpp
template<class I>
concept no-throw-input-iterator = // exposition only
    input_iterator<I> &&
    is_lvalue_reference_v<iter_reference_t<I>> &&
    same_as<remove_cvref_t<iter_reference_t<I>>, iter_value_t<I>>;
```

A type I models no-throw-input-iterator only if no exceptions are thrown from increment, copy construction, move construction, copy assignment, move assignment, or indirection through valid iterators.

[Note 1: This concept allows some input_iterator (23.3.4.9) operations to throw exceptions. — end note]

```cpp
template<class S, class I>
concept no-throw-sentinel-for = sentinel_for<S, I>; // exposition only
```

Types S and I model no-throw-sentinel-for only if no exceptions are thrown from copy construction, move construction, copy assignment, move assignment, or comparisons between valid values of type I and S.

[Note 2: This concept allows some sentinel_for (23.3.4.7) operations to throw exceptions. — end note]
template<class R>
concept no-throw-input-range = // exposition only
range<R> &&
no-throw-input-iterator<iterator_t<R>> &&
no-throw-sentinel-for<sentinel_t<R>, iterator_t<R>>;

A type R models no-throw-input-range only if no exceptions are thrown from calls to ranges::begin and ranges::end on an object of type R.

template<class I>
concept no-throw-forward-iterator = // exposition only
no-throw-input-iterator<I> &&
forward_iterator<I> &&
no-throw-sentinel-for<I, I>;

[Note 3: This concept allows some forward_iterator (23.3.4.11) operations to throw exceptions. — end note]

template<class R>
concept no-throw-forward-range = // exposition only
no-throw-input-range<R> &&
no-throw-forward-iterator<iterator_t<R>>;

25.11.3 uninitialized_default_construct [uninitialized.construct.default]

template<class NoThrowForwardIterator>
void uninitialized_default_construct(NoThrowForwardIterator first, NoThrowForwardIterator last);

Effects: Equivalent to:
for (; first != last; ++first)
::new (voidify(*first))
typename iterator_traits<NoThrowForwardIterator>::value_type;

namespace ranges {
    template<no-throw-forward-iterator I, no-throw-sentinel-for<I> S>
    requires default_initializable<iter_value_t<I>>
    I uninitialized_default_construct(I first, S last);
    template<no-throw-forward-range R>
    requires default_initializable<range_value_t<R>>
    borrowed_iterator_t<R> uninitialized_default_construct(R&& r);
}

Effects: Equivalent to:
for (; first != last; ++first)
::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>;
return first;

template<class NoThrowForwardIterator, class Size>
NoThrowForwardIterator uninitialized_default_construct_n(NoThrowForwardIterator first, Size n);

Effects: Equivalent to:
for (; n > 0; (void)++first, --n)
::new (voidify(*first))
typename iterator_traits<NoThrowForwardIterator>::value_type;
return first;

namespace ranges {
    template<no-throw-forward-iterator I>
    requires default_initializable<iter_value_t<I>>
    I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
}

Effects: Equivalent to:
return uninitialized_default_construct(counted_iterator(first, n),
default_sentinel).base();
25.11.4 uninitialized_value_construct

```
template<class NoThrowForwardIterator>
void uninitialized_value_construct(NoThrowForwardIterator first, NoThrowForwardIterator last);
```

1. **Effects**: Equivalent to:
   ```
   for (; first != last; ++first)
       ::new (voidify(*first))
           typename iterator_traits<NoThrowForwardIterator>::value_type();
   ```

namespace ranges {
    template<no-throw-forward_iterator I, no-throw-sentinel-for<I> S>
        requires default_initializable<iter_value_t<I>>
    I uninitialized_value_construct(I first, S last);
    template<no-throw-forward-range R>
        requires default_initializable<range_value_t<R>>
    borrowed_iterator_t<R> uninitialized_value_construct(R&& r);
}

2. **Effects**: Equivalent to:
   ```
   for (; first != last; ++first)
       ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>();
   ```

   return first;

    template<class NoThrowForwardIterator, class Size>
    NoThrowForwardIterator uninitialized_value_construct_n(NoThrowForwardIterator first, Size n);

3. **Effects**: Equivalent to:
   ```
   for (; n > 0; (void)++first, --n)
       ::new (voidify(*first))
           typename iterator_traits<NoThrowForwardIterator>::value_type();
   ```

4. **Returns**: result.

namespace ranges {
    template<no-throw-forward_iterator I>
        requires default_initializable<iter_value_t<I>>
    I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
}

25.11.5 uninitialized_copy

```
template<class InputIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_copy(InputIterator first, InputIterator last, NoThrowForwardIterator result);
```

1. **Preconditions**: result + [0, (last - first)) does not overlap with [first, last).

2. **Effects**: Equivalent to:
   ```
   for (; first != last; ++result, (void)++first)
       ::new (voidify(*result))
           typename iterator_traits<NoThrowForwardIterator>::value_type(*first);
   ```

3. **Returns**: result.

namespace ranges {
    template<input_iterator I, sentinel_for<I> S1, no-throw-forward_iterator O, no-throw-sentinel-for<O> S2>
        requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
    uninitialized_copy_result<I, O>
    uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);
}
template<input_range IR, no-throw-forward-range OR>>
uninitialized_copy_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
uninitialized_copy(IR&& in_range, OR&& out_range);
}

4 Preconditions: [ofirst, olast) does not overlap with [ifirst, ilast).
5 Effects: Equivalent to:
   for (; ifirst != ilast && ofirst != olast; ++ofirst, (void) ++ifirst)
     ::new (voidify(*ofirst)) remove_reference_t<iter_reference_t<O>>(*ifirst);
return {std::move(ifirst), ofirst};

template<class InputIterator, class Size, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_copy_n(InputIterator first, Size n,
NoThrowForwardIterator result);

6 Preconditions: result + [0, n) does not overlap with first + [0, n).
7 Effects: Equivalent to:
   for ( ; n > 0; ++result, (void) ++first, --n)
     ::new (voidify(*result))
     typename iterator_traits<NoThrowForwardIterator>::value_type(*first);
8 Returns: result.

namespace ranges {
    template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel-for<O> S>
    requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
    uninitialized_copy_n_result<I, O>
    uninitialized_copy_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}

9 Preconditions: [ofirst, olast) does not overlap with ifirst + [0, n).
10 Effects: Equivalent to:
     auto t = uninitialized_copy(counted_iterator/ifirst, n),
     default_sentinel, ofirst, olast);
     return {std::move(t.in).base(), t.out};

25.11.6 uninitialized_move

[uninitialized.move]

template<class InputIterator, class NoThrowForwardIterator>
NoThrowForwardIterator uninitialized_move(InputIterator first, InputIterator last,
NoThrowForwardIterator result);

1 Preconditions: result + [0, (last - first)) does not overlap with [first, last).
2 Effects: Equivalent to:
   for (; first != last; (void)++result, ++first)
     ::new (voidify(*result))
     typename iterator_traits<NoThrowForwardIterator>::value_type(std::move(*first));
return result;

namespace ranges {
    template<input_iterator I, sentinel_for<I> S1, no-throw-forward-iterator O, no-throw-sentinel-for<O> S2>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_result<I, O>
    uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
    template<input_range IR, no-throw-forward-range OR>
    requires constructible_from<range_value_t<OR>, range_rvalue_reference_t<IR>>
    uninitialized_move_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
    uninitialized_move(IR&& in_range, OR&& out_range);
}

3 Preconditions: [ofirst, olast) does not overlap with [ifirst, ilast).
4 Effects: Equivalent to:
for (; ifirst != ilast && ofirst != olast; ++ofirst, (void) ++ifirst)
  :new (voidify(*ofirst))
  remove_reference_t<iter_reference_t<O>>(ranges::iter_move(ifirst));
return {std::move(ifirst), ofirst};

[Note 1: If an exception is thrown, some objects in the range [first, last) are left in a valid, but unspecified state. — end note]

template<class InputIterator, class Size, class NoThrowForwardIterator>
pair<InputIterator, NoThrowForwardIterator>
uninitialized_move_n(InputIterator first, Size n, NoThrowForwardIterator result);

Preconditions: result + [0, n) does not overlap with first + [0, n).

Effects: Equivalent to:
  for (; n > 0; ++result, (void) ++first, --n)
  :new (voidify(*result))
  typename iterator_traits<NoThrowForwardIterator>::value_type(std::move(*first));
return {first, result};

namespace ranges {
  template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel-for<O> S>
  requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
  uninitialized_move_n_result<I, O> uninitialized_move_n(I ifirst, iter_difference_t<I> n, 0 ofirst, S olast);
}

Preconditions: [ofirst, olast) does not overlap with ifirst + [0, n).

Effects: Equivalent to:
  auto t = uninitialized_move(counted_iterator(ifirst, n),
  default_sentinel, ofirst, olast);
  return {std::move(t.in).base(), t.out};

[Note 2: If an exception is thrown, some objects in the range first + [0, n) are left in a valid but unspecified state. — end note]

25.11.7 uninitialized_fill

template<class NoThrowForwardIterator, class T>
void uninitialized_fill(NoThrowForwardIterator first, NoThrowForwardIterator last, const T& x);

Effects: Equivalent to:
  for (; first != last; ++first)
  :new (voidify(*first))
  typename iterator_traits<NoThrowForwardIterator>::value_type(x);

namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel-for<I> S, class T>
  requires constructible_from<iter_value_t<I>, const T&>
  I uninitialized_fill(I first, S last, const T& x);
  template<no-throw-forward-range R, class T>
  requires constructible_from<range_value_t<R>, const T&>
  borrowed_iterator_t<R> uninitialized_fill(R&& r, const T& x);
}

Effects: Equivalent to:
  for (; first != last; ++first)
  :new (voidify(*first))
  remove_reference_t<iter_reference_t<I>>(x);
return first;

template<class NoThrowForwardIterator, class Size, class T>
NoThrowForwardIterator uninitialized_fill_n(NoThrowForwardIterator first, Size n, const T& x);

Effects: Equivalent to:
  for (; n--; ++first)
  :new (voidify(*first))
  typename iterator_traits<NoThrowForwardIterator>::value_type(x);
return first;

namespace ranges {
  template<no-throw-forward-iterator I, class T>
  requires constructible_from<iter_value_t<I>, const T&>
  I uninitialized_fill_n(I first, iter_difference_t<I> n, const T& x);
}

Effects: Equivalent to:

return uninitialized_fill(counted_iterator(first, n), default_sentinel, x).base();

25.11.8 construct_at

[specialized.construct]

template<class T, class... Args>
constexpr T* construct_at(T* location, Args&&... args);

namespace ranges {
  template<class T, class... Args>
  constexpr T* construct_at(T* location, Args&&... args);
}

Constraints: The expression ::new (declval<void*>(())) T(declval<Args>(...)...) is well-formed when treated as an unevaluated operand.

Effects: Equivalent to:

return ::new (voidify(*location)) T(std::forward<Args>(args)...);

25.11.9 destroy

[specialized.destroy]

template<class T>
constexpr void destroy_at(T* location);

namespace ranges {
  template<destructible T>
  constexpr void destroy_at(T* location) noexcept;
}

Effects:

(1.1) If T is an array type, equivalent to destroy(begin(*location), end(*location)).

(1.2) Otherwise, equivalent to location->~T().

template<class NoThrowForwardIterator>
constexpr void destroy(NoThrowForwardIterator first, NoThrowForwardIterator last);

Effects: Equivalent to:

for (; first != last; ++first)
  destroy_at(addressof(*first));

namespace ranges {
  template<no-throw-input-iterator I, no-throw-sentinel_for<I> S>
  requires destructible<iter_value_t<I>>
  constexpr I destroy(I first, S last) noexcept;
  template<no-throw-input-range R>
  requires destructible<range_value_t<R>>
  constexpr borrowed_iterator_t<R> destroy(R&& r) noexcept;
}

Effects: Equivalent to:

for (; first != last; ++first)
  destroy_at(addressof(*first));
  return first;

template<class NoThrowForwardIterator, class Size>
constexpr NoThrowForwardIterator destroy_n(NoThrowForwardIterator first, Size n);

Effects: Equivalent to:

§ 25.11.9
for (; n > 0; (void)++first, --n)
    destroy_at(addressof(*first));
return first;

namespace ranges {
    template<no-throw-input-iterator I>
    requires destructible<iter_value_t<I>>
    constexpr I destroy_n(I first, iter_difference_t<I> n) noexcept;
}

Effects: Equivalent to:
    return destroy(counted_iterator(first, n), default_sentinel).base();

25.12 C library algorithms [alg.c.library]

1 [Note 1: The header <cstdlib> (17.2.2) declares the functions described in this subclause. — end note]

    void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
                   c-compare-pred* compar);
    void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
                   compare-pred* compar);
    void qsort(void* base, size_t nmemb, size_t size, c-compare-pred* compar);
    void qsort(void* base, size_t nmemb, size_t size, compare-pred* compar);

2 Preconditions: The objects in the array pointed to by base are of trivial type.

3 Effects: These functions have the semantics specified in the C standard library.

4 Throws: Any exception thrown by compar (16.4.6.13).

See also: ISO C 7.22.5.
26 Numerics library

26.1 General

This Clause describes components that C++ programs may use to perform seminumerical operations.

The following subclauses describe components for complex number types, random number generation, numeric (n-at-a-time) arrays, generalized numeric algorithms, and mathematical constants and functions for floating-point types, as summarized in Table 92.

<table>
<thead>
<tr>
<th>Table 92: Numerics library summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclause Header</td>
</tr>
<tr>
<td>26.2 Requirements</td>
</tr>
<tr>
<td>26.3 Floating-point environment</td>
</tr>
<tr>
<td>26.4 Complex numbers</td>
</tr>
<tr>
<td>26.5 Bit manipulation</td>
</tr>
<tr>
<td>26.6 Random number generation</td>
</tr>
<tr>
<td>26.7 Numeric arrays</td>
</tr>
<tr>
<td>26.8 Mathematical functions for floating-point types</td>
</tr>
<tr>
<td>26.9 Numbers</td>
</tr>
</tbody>
</table>

26.2 Numeric type requirements

The complex and valarray components are parameterized by the type of information they contain and manipulate. A C++ program shall instantiate these components only with a numeric type. A numeric type is a cv-unqualified object type T that meets the Cpp17DefaultConstructible, Cpp17CopyConstructible, Cpp17CopyAssignable, and Cpp17Destructible requirements (16.4.4.2).

If any operation on T throws an exception the effects are undefined.

In addition, many member and related functions of valarray<T> can be successfully instantiated and will exhibit well-defined behavior if and only if T meets additional requirements specified for each such member or related function.

Example 1: It is valid to instantiate valarray<complex>, but operator() will not be successfully instantiated for valarray<complex> operands, since complex does not have any ordering operators. —end example

26.3 The floating-point environment

26.3.1 Header <cfenv>

#define FE_ALL_EXCEPT see below
#define FE_DDIVBYZERO see below    // optional
#define FE_INEXACT see below       // optional
#define FE_INVALID see below       // optional
#define FE_OVERFLOW see below      // optional
#define FE_UNDERFLOW see below     // optional
#define FE_DOWNWARD see below      // optional
#define FE_TONEAREST see below     // optional
#define FE_TOWARDZERO see below    // optional
#define FE_UPWARD see below        // optional
#define FE_DFL_ENV see below
namespace std {
    // types
    using fenv_t   = object type;
    using fexcept_t = integer type;

    // functions
    int feclearexcept(int except);
    int fegetexceptflag(fexcept_t* pflag, int except);
    int feraiseexcept(int except);
    int fesetexceptflag(const fexcept_t* pflag, int except);
    int fetestexcept(int except);

    int fegetround();
    int fesetround(int mode);

    int fegetenv(fenv_t* penv);
    int feholdexcept(fenv_t* penv);
    int fesetenv(const fenv_t* penv);
    int feupdateenv(const fenv_t* penv);
}

1 The contents and meaning of the header <cfenv> are the same as the C standard library header <fenv.h>. [Note 1: This document does not require an implementation to support the FENV_ACCESS pragma; it is implementation-defined (15.9) whether the pragma is supported. As a consequence, it is implementation-defined whether these functions can be used to test floating-point status flags, set floating-point control modes, or run under non-default mode settings. If the pragma is used to enable control over the floating-point environment, this document does not specify the effect on floating-point evaluation in constant expressions. — end note]

SEE ALSO: ISO C 7.6

26.3.2 Threads

The floating-point environment has thread storage duration (6.7.5.3). The initial state for a thread’s floating-point environment is the state of the floating-point environment of the thread that constructs the corresponding thread object (32.4.3) or jthread object (32.4.4) at the time it constructed the object. [Note 1: That is, the child thread gets the floating-point state of the parent thread at the time of the child’s creation. — end note]

2 A separate floating-point environment is maintained for each thread. Each function accesses the environment corresponding to its calling thread.

26.4 Complex numbers

26.4.1 General

The header <complex> defines a class template, and numerous functions for representing and manipulating complex numbers.

The effect of instantiating the template complex for any type other than float, double, or long double is unspecified. The specializations complex<float>, complex<double>, and complex<long double> are literal types (6.8).

If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

If z is an lvalue of type cv complex<T> then:

(4.1) — the expression reinterpret_cast<cv T(&)[2]>(z) is well-formed,

(4.2) — reinterpret_cast<cv T(&)[2]>(z)[0] designates the real part of z, and

(4.3) — reinterpret_cast<cv T(&)[2]>(z)[1] designates the imaginary part of z.

Moreover, if a is an expression of type cv complex<T>* and the expression a[i] is well-defined for an integer expression i, then:

(4.4) — reinterpret_cast<cv T*>(a)[2*i] designates the real part of a[i], and

(4.5) — reinterpret_cast<cv T*>(a)[2*i + 1] designates the imaginary part of a[i].
26.4.2 Header <complex> synopsis

namespace std {
    // 26.4.3, class template complex
    template<class T> class complex;

    // 26.4.4, specializations
    template<> class complex<float>;
    template<> class complex<double>;
    template<> class complex<long double>;

    // 26.4.7, operators
    template<class T> constexpr complex<T> operator+(const complex<T>&, const complex<T>&);
    template<class T> constexpr complex<T> operator+(const complex<T>&, const T&);
    template<class T> constexpr complex<T> operator+(const T&, const complex<T>&);
    template<class T> constexpr complex<T> operator-(const complex<T>&, const complex<T>&);
    template<class T> constexpr complex<T> operator-(const complex<T>&, const T&);
    template<class T> constexpr complex<T> operator-(const T&, const complex<T>&);
    template<class T> constexpr complex<T> operator*(const complex<T>&, const complex<T>&);
    template<class T> constexpr complex<T> operator*(const complex<T>&, const T&);
    template<class T> constexpr complex<T> operator*(const T&, const complex<T>&);
    template<class T> constexpr complex<T> operator/(const complex<T>&, const complex<T>&);
    template<class T> constexpr complex<T> operator/(const complex<T>&, const T&);
    template<class T> constexpr complex<T> operator/(const T&, const complex<T>&);
    template<class T> constexpr complex<T> operator+(const complex<T>&);
    template<class T> constexpr complex<T> operator-(const complex<T>&);
    template<class T, class charT, class traits> basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&, complex<T>&);
    template<class T, class charT, class traits> basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, const complex<T>&);

    // 26.4.8, values
    template<class T> constexpr T real(const complex<T>&);
    template<class T> constexpr T imag(const complex<T>&);
    template<class T> T abs(const complex<T>&);
    template<class T> T arg(const complex<T>&);
    template<class T> constexpr T norm(const complex<T>&);
    template<class T> constexpr complex<T> conj(const complex<T>&);
    template<class T> complex<T> proj(const complex<T>&);
    template<class T> complex<T> polar(const T&, const T& = T());

    // 26.4.9, transcendents
    template<class T> complex<T> acos(const complex<T>&);
    template<class T> complex<T> asin(const complex<T>&);
    template<class T> complex<T> atan(const complex<T>&);
    template<class T> complex<T> acosh(const complex<T>&);
    template<class T> complex<T> asinh(const complex<T>&);
    template<class T> complex<T> atanh(const complex<T>&);
    template<class T> complex<T> cos(const complex<T>&);
    template<class T> complex<T> cosh(const complex<T>&);
    template<class T> complex<T> exp(const complex<T>&);
}
template<class T> complex<T> log (const complex<T>&);
template<class T> complex<T> log10(const complex<T>&);

template<class T> complex<T> pow (const complex<T>&, const T&);
template<class T> complex<T> pow (const complex<T>&, const complex<T>&);
template<class T> complex<T> pow (const T&, const complex<T>&);

template<class T> complex<T> sin (const complex<T>&);
template<class T> complex<T> sinh (const complex<T>&);
template<class T> complex<T> sqrt (const complex<T>&);
template<class T> complex<T> tan (const complex<T>&);
template<class T> complex<T> tanh (const complex<T>&);

// 26.4.11, complex literals
inline namespace literals {
    inline namespace complex_literals {
        constexpr complex<long double> operator"il(long double);
        constexpr complex<long double> operator"il(unsigned long long);
        constexpr complex<double> operator"i(long double);
        constexpr complex<double> operator"i(unsigned long long);
        constexpr complex<float> operator"if(long double);
        constexpr complex<float> operator"if(unsigned long long);
    }
}

26.4.3 Class template complex

namespace std {
    template<class T> class complex {
    public:
        using value_type = T;

        constexpr complex(const T& re = T(), const T& im = T());
        constexpr complex(const complex&);
        template<class X> constexpr complex(const complex<X>&);

        constexpr T real() const;
        constexpr void real(T);
        constexpr T imag() const;
        constexpr void imag(T);

        constexpr complex& operator= (const T&);
        constexpr complex& operator+=(const T&);
        constexpr complex& operator-=(const T&);
        constexpr complex& operator*=(const T&);
        constexpr complex& operator/=(const T&);
        template<class X> constexpr complex& operator= (const complex<X>&);
        template<class X> constexpr complex& operator+=(const complex<X>&);
        template<class X> constexpr complex& operator-=(const complex<X>&);
        template<class X> constexpr complex& operator*=(const complex<X>&);
        template<class X> constexpr complex& operator/=(const complex<X>&);
    }
};

1 The class complex describes an object that can store the Cartesian components, real() and imag(), of a complex number.

26.4.4 Specializations

namespace std {
    template<> class complex<float> {
    public:
        using value_type = float;
    }
}
constexpr complex(float re = 0.0f, float im = 0.0f);
constexpr complex(const complex<float>&) = default;
constexpr explicit complex(const complex<double>&);
constexpr explicit complex(const complex<long double>&);

constexpr float real() const;
constexpr void real(float);
constexpr float imag() const;
constexpr void imag(float);

constexpr complex& operator= (float);
constexpr complex& operator+=(float);
constexpr complex& operator-=(float);
constexpr complex& operator*=(float);
constexpr complex& operator/=(float);

constexpr complex& operator=(const complex&);
template<class X> constexpr complex& operator= (const complex<X>&);
}

template<> class complex<double> {
public:
  using value_type = double;

  constexpr complex(double re = 0.0, double im = 0.0);
  constexpr complex(const complex<float>&);
  constexpr complex(const complex<double>&) = default;
  constexpr explicit complex(const complex<long double>&);

  constexpr double real() const;
  constexpr void real(double);
  constexpr double imag() const;
  constexpr void imag(double);

  constexpr complex& operator= (double);
  constexpr complex& operator+=(double);
  constexpr complex& operator-=(double);
  constexpr complex& operator*=(double);
  constexpr complex& operator/=(double);

  constexpr complex& operator=(const complex&);
template<class X> constexpr complex& operator= (const complex<X>&);
};

template<> class complex<long double> {
public:
  using value_type = long double;

  constexpr complex(long double re = 0.0L, long double im = 0.0L);
  constexpr complex(const complex<float>&);
  constexpr complex(const complex<double>&);
  constexpr complex(const complex<long double>&) = default;

  constexpr long double real() const;
  constexpr void real(long double);
  constexpr long double imag() const;
  constexpr void imag(long double);

```
constexpr void imag(long double);
constexpr complex& operator= (long double);
constexpr complex& operator+=(long double);
constexpr complex& operator-=(long double);
constexpr complex& operator*=(long double);
constexpr complex& operator/=(long double);
constexpr complex& operator=(const complex&);
template<class X> constexpr complex& operator= (const complex<X>&);
template<class X> constexpr complex& operator+=(const complex<X>&);
template<class X> constexpr complex& operator-=(const complex<X>&);
template<class X> constexpr complex& operator*=(const complex<X>&);
template<class X> constexpr complex& operator/=(const complex<X>&);
};

26.4.5 Member functions

template<class T> constexpr complex(const T& re = T(), const T& im = T());

1 Postconditions: \( \text{real}() = re \) && \( \text{imag}() = im \) is true.

constexpr T real() const;
2 Returns: The value of the real component.

constexpr void real(T val);
3 Effects: Assigns \( val \) to the real component.

constexpr T imag() const;
4 Returns: The value of the imaginary component.

constexpr void imag(T val);
5 Effects: Assigns \( val \) to the imaginary component.

26.4.6 Member operators

constexpr complex& operator+=(const T& rhs);
1 Effects: Adds the scalar value \( \text{rhs} \) to the real part of the complex value \(*this\) and stores the result in the real part of \(*this\), leaving the imaginary part unchanged.

2 Returns: \(*this\).

constexpr complex& operator-=(const T& rhs);
3 Effects: Subtracts the scalar value \( \text{rhs} \) from the real part of the complex value \(*this\) and stores the result in the real part of \(*this\), leaving the imaginary part unchanged.

4 Returns: \(*this\).

constexpr complex& operator*=(const T& rhs);
5 Effects: Multiplies the scalar value \( \text{rhs} \) by the complex value \(*this\) and stores the result in \(*this\).

6 Returns: \(*this\).

constexpr complex& operator/=(const T& rhs);
7 Effects: Divides the scalar value \( \text{rhs} \) into the complex value \(*this\) and stores the result in \(*this\).

8 Returns: \(*this\).

template<class X> constexpr complex& operator+=(const complex<X>& rhs);
9 Effects: Adds the complex value \( \text{rhs} \) to the complex value \(*this\) and stores the sum in \(*this\).

10 Returns: \(*this\).
template<class X> constexpr complex& operator-=(const complex<X>& rhs);

**Effects**: Subtracts the complex value \( \text{rhs} \) from the complex value \(*\text{this}\) and stores the difference in \(*\text{this}\).

**Returns**: \(*\text{this}\).

template<class X> constexpr complex& operator*=(const complex<X>& rhs);

**Effects**: Multiplies the complex value \( \text{rhs} \) by the complex value \(*\text{this}\) and stores the product in \(*\text{this}\).

**Returns**: \(*\text{this}\).

template<class X> constexpr complex& operator/=(const complex<X>& rhs);

**Effects**: Divides the complex value \( \text{rhs} \) into the complex value \(*\text{this}\) and stores the quotient in \(*\text{this}\).

**Returns**: \(*\text{this}\).

### 26.4.7 Non-member operations

[**complex.ops**]

template<class T> constexpr complex<T> operator+(const complex<T>& lhs);

**Returns**: \( \text{complex}(\text{lhs}) \).

template<class T> constexpr complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);

template<class T> constexpr complex<T> operator+(const complex<T>& lhs, const T& rhs);

template<class T> constexpr complex<T> operator+(const T& lhs, const complex<T>& rhs);

**Returns**: \( \text{complex}(\text{lhs}) + \text{rhs} \).

template<class T> constexpr complex<T> operator-(const complex<T>& lhs);

template<class T> constexpr complex<T> operator-(const complex<T>& lhs, const complex<T>& rhs);

template<class T> constexpr complex<T> operator-(const complex<T>& lhs, const T& rhs);

template<class T> constexpr complex<T> operator-(const T& lhs, const complex<T>& rhs);

**Returns**: \( \text{complex}(\text{lhs}) - \text{rhs} \).

template<class T> constexpr complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);

template<class T> constexpr complex<T> operator*(const complex<T>& lhs, const T& rhs);

template<class T> constexpr complex<T> operator*(const T& lhs, const complex<T>& rhs);

**Returns**: \( \text{complex}(\text{lhs}) \times \text{rhs} \).

template<class T> constexpr complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);

template<class T> constexpr complex<T> operator/(const complex<T>& lhs, const T& rhs);

template<class T> constexpr complex<T> operator/(const T& lhs, const complex<T>& rhs);

**Returns**: \( \text{complex}(\text{lhs}) \div \text{rhs} \).

template<class T> constexpr bool operator==(const complex<T>& lhs, const complex<T>& rhs);

template<class T> constexpr bool operator==(const complex<T>& lhs, const T& rhs);

**Returns**: \( \text{lhs.real()} = \text{rhs.real()} \) && \( \text{lhs.imag()} = \text{rhs.imag()} \).

**Remarks**: The imaginary part is assumed to be \( T() \), or 0.0, for the \( T \) arguments.

### 26.4.7.26.4.7.2 § 26.4.7 1167

template<class T, class charT, class traits>

basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& is, complex<T>& x);

**Preconditions**: The input values are convertible to \( T \).

**Effects**: Extracts a complex number \( x \) of the form: \( u, (u,v), \) or \( (u,v) \), where \( u \) is the real part and \( v \) is the imaginary part (29.7.4.3).

If bad input is encountered, calls \( \text{is.setstate(ios_base::failbit)} \) (which may throw \( \text{ios_base::failure}(29.5.5.4) \)).

**Returns**: \( \text{is} \).

**Remarks**: This extraction is performed as a series of simpler extractions. Therefore, the skipping of whitespace is specified to be the same for each of the simpler extractions.
N4868

© ISO/IEC

template<class T, class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);
14

Effects: Inserts the complex number x onto the stream o as if it were implemented as follows:
basic_ostringstream<charT, traits> s;
s.flags(o.flags());
s.imbue(o.getloc());
s.precision(o.precision());
s << ’(’ << x.real() << "," << x.imag() << ’)’;
return o << s.str();

15

[Note 1 : In a locale in which comma is used as a decimal point character, the use of comma as a field separator
can be ambiguous. Inserting showpoint into the output stream forces all outputs to show an explicit decimal
point character; as a result, all inserted sequences of complex numbers can be extracted unambiguously. — end
note]

26.4.8

Value operations

[complex.value.ops]

template<class T> constexpr T real(const complex<T>& x);
1

Returns: x.real().
template<class T> constexpr T imag(const complex<T>& x);

2

Returns: x.imag().
template<class T> T abs(const complex<T>& x);

3

Returns: The magnitude of x.
template<class T> T arg(const complex<T>& x);

4

Returns: The phase angle of x, or atan2(imag(x), real(x)).
template<class T> constexpr T norm(const complex<T>& x);

5

Returns: The squared magnitude of x.
template<class T> constexpr complex<T> conj(const complex<T>& x);

6

Returns: The complex conjugate of x.
template<class T> complex<T> proj(const complex<T>& x);

7

Returns: The projection of x onto the Riemann sphere.

8

Remarks: Behaves the same as the C function cproj. See also: ISO C 7.3.9.5
template<class T> complex<T> polar(const T& rho, const T& theta = T());

9
10

Preconditions: rho is non-negative and non-NaN. theta is finite.
Returns: The complex value corresponding to a complex number whose magnitude is rho and whose
phase angle is theta.

26.4.9

Transcendentals

[complex.transcendentals]

template<class T> complex<T> acos(const complex<T>& x);
1

Returns: The complex arc cosine of x.

2

Remarks: Behaves the same as the C function cacos. See also: ISO C 7.3.5.1
template<class T> complex<T> asin(const complex<T>& x);

3

Returns: The complex arc sine of x.

4

Remarks: Behaves the same as the C function casin. See also: ISO C 7.3.5.2
template<class T> complex<T> atan(const complex<T>& x);

5

Returns: The complex arc tangent of x.

6

Remarks: Behaves the same as the C function catan. See also: ISO C 7.3.5.3

§ 26.4.9

1168


template<class T> complex<T> acosh(const complex<T>& x);

Returns: The complex arc hyperbolic cosine of \( x \).

Remarks: Behaves the same as the C function \( \text{cacosh} \). See also: ISO C 7.3.6.1

template<class T> complex<T> asinh(const complex<T>& x);

Returns: The complex arc hyperbolic sine of \( x \).

Remarks: Behaves the same as the C function \( \text{casinh} \). See also: ISO C 7.3.6.2

template<class T> complex<T> atanh(const complex<T>& x);

Returns: The complex arc hyperbolic tangent of \( x \).

Remarks: Behaves the same as the C function \( \text{catanh} \). See also: ISO C 7.3.6.3

template<class T> complex<T> cos(const complex<T>& x);

Returns: The complex cosine of \( x \).

template<class T> complex<T> cosh(const complex<T>& x);

Returns: The complex hyperbolic cosine of \( x \).

template<class T> complex<T> exp(const complex<T>& x);

Returns: The complex base-\( e \) exponential of \( x \).

template<class T> complex<T> log(const complex<T>& x);

Returns: The complex natural (base-\( e \)) logarithm of \( x \). For all \( x \), \( \text{imag}(\log(x)) \) lies in the interval \( [-\pi, \pi] \).

[Note 1: The semantics of this function are intended to be the same in C++ as they are for \( \text{clog} \) in C. — end note]

Remarks: The branch cuts are along the negative real axis.

template<class T> complex<T> log10(const complex<T>& x);

Returns: The complex common (base-10) logarithm of \( x \), defined as \( \log(x) / \log(10) \).

Remarks: The branch cuts are along the negative real axis.

template<class T> complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> pow(const complex<T>& x, const T& y);
template<class T> complex<T> pow(const T& x, const complex<T>& y);

Returns: The complex power of base \( x \) raised to the \( y \)th power, defined as \( \exp(y \times \log(x)) \). The value returned for \( \text{pow}(0, 0) \) is implementation-defined.

Remarks: The branch cuts are along the negative real axis.

template<class T> complex<T> sin(const complex<T>& x);

Returns: The complex sine of \( x \).

template<class T> complex<T> sinh(const complex<T>& x);

Returns: The complex hyperbolic sine of \( x \).

template<class T> complex<T> sqrt(const complex<T>& x);

Returns: The complex square root of \( x \), in the range of the right half-plane.

[Note 2: The semantics of this function are intended to be the same in C++ as they are for \( \text{csqrt} \) in C. — end note]

Remarks: The branch cuts are along the negative real axis.

template<class T> complex<T> tan(const complex<T>& x);

Returns: The complex tangent of \( x \).

template<class T> complex<T> tanh(const complex<T>& x);

Returns: The complex hyperbolic tangent of \( x \).
26.4.10 Additional overloads

The following function templates shall have additional overloads:

arg norm
conj proj
imag real

where norm, conj, imag, and real are constexpr overloads.

2 The additional overloads shall be sufficient to ensure:

(2.1) — If the argument has type long double, then it is effectively cast to complex<long double>.
(2.2) — Otherwise, if the argument has type double or an integer type, then it is effectively cast to complex<double>.
(2.3) — Otherwise, if the argument has type float, then it is effectively cast to complex<float>.

Function template pow shall have additional overloads sufficient to ensure, for a call with at least one argument of type complex<T>:

(3.1) — If either argument has type complex<long double> or type long double, then both arguments are effectively cast to complex<long double>.
(3.2) — Otherwise, if either argument has type complex<double>, double, or an integer type, then both arguments are effectively cast to complex<double>.
(3.3) — Otherwise, if either argument has type complex<float> or float, then both arguments are effectively cast to complex<float>.

26.4.11 Suffixes for complex number literals

This subclause describes literal suffixes for constructing complex number literals. The suffixes i, il, and if create complex numbers of the types complex<double>, complex<long double>, and complex<float> respectively, with their imaginary part denoted by the given literal number and the real part being zero.

constexpr complex<long double> operator""il(long double d);
constexpr complex<long double> operator""il(unsigned long long d);

Returns: complex<long double>{0.0L, static_cast<long double>(d)}.

constexpr complex<double> operator""i(long double d);
constexpr complex<double> operator""i(unsigned long long d);

Returns: complex<double>{0.0, static_cast<double>(d)}.

constexpr complex<float> operator""if(long double d);
constexpr complex<float> operator""if(unsigned long long d);

Returns: complex<float>{0.0f, static_cast<float>(d)}.

26.5 Bit manipulation

26.5.1 General

The header <bit> provides components to access, manipulate and process both individual bits and bit sequences.

26.5.2 Header <bit> synopsis

namespace std {

// 26.5.3, bit_cast
template<class To, class From>
constexpr To bit_cast(const From& from) noexcept;

// 26.5.4, integral powers of 2
template<class T>
constexpr bool has_single_bit(T x) noexcept;
template<class T>
constexpr T bit_ceil(T x);
template<class T>
constexpr T bit_floor(T x) noexcept;

§ 26.5.2 1170
template<class T>
constexpr T bit_width(T x) noexcept;

// 26.5.5, rotating
template<class T>
[[nodiscard]] constexpr T rotl(T x, int s) noexcept;
template<class T>
[[nodiscard]] constexpr T rotr(T x, int s) noexcept;

// 26.5.6, counting
template<class T>
constexpr int countl_zero(T x) noexcept;
template<class T>
constexpr int countl_one(T x) noexcept;
template<class T>
constexpr int countr_zero(T x) noexcept;
template<class T>
constexpr int countr_one(T x) noexcept;
template<class T>
constexpr int popcount(T x) noexcept;

// 26.5.7, endian
enum class endian {
    little = see below,
    big = see below,
    native = see below
};

26.5.3 Function template bit_cast

template<class To, class From>
constexpr To bit_cast(const From& from) noexcept;

1 Constraints:
(1.1) sizeof(To) == sizeof(From) is true;
(1.2) is_trivially_copyable_v<To> is true; and
(1.3) is_trivially_copyable_v<From> is true.

2 Returns: An object of type To. Implicitly creates objects nested within the result (6.7.2). Each bit of the value representation of the result is equal to the corresponding bit in the object representation of from. Padding bits of the result are unspecified. For the result and each object created within it, if there is no value of the object's type corresponding to the value representation produced, the behavior is undefined. If there are multiple such values, which value is produced is unspecified.

3 Remarks: This function is constexpr if and only if To, From, and the types of all subobjects of To and From are types T such that:
(3.1) is_union_v<T> is false;
(3.2) is_pointer_v<T> is false;
(3.3) is_member_pointer_v<T> is false;
(3.4) is_volatile_v<T> is false; and
(3.5) T has no non-static data members of reference type.

26.5.4 Integral powers of 2

template<class T>
constexpr bool has_single_bit(T x) noexcept;

1 Constraints: T is an unsigned integer type (6.8.2).

2 Returns: true if x is an integral power of two; false otherwise.
template<class T>
 constexpr T bit_ceil(T x);

Let \( N \) be the smallest power of 2 greater than or equal to \( x \).

**Constraints:** \( T \) is an unsigned integer type (6.8.2).

**Preconditions:** \( N \) is representable as a value of type \( T \).

**Returns:** \( N \).

**Throws:** Nothing.

**Remarks:** A function call expression that violates the precondition in the **Preconditions** element is not a core constant expression (7.7).

template<class T>
 constexpr T bit_floor(T x) noexcept;

**Constraints:** \( T \) is an unsigned integer type (6.8.2).

**Returns:** If \( x == 0 \), 0; otherwise the maximal value \( y \) such that `has_single_bit(y)` is true and \( y <= x \).

template<class T>
 constexpr T bit_width(T x) noexcept;

**Constraints:** \( T \) is an unsigned integer type (6.8.2).

**Returns:** If \( x == 0 \), 0; otherwise one plus the base-2 logarithm of \( x \), with any fractional part discarded.

### 26.5.5 Rotating

In the following descriptions, let \( N \) denote `numeric_limits<T>::digits`.

```cpp
template<class T>
[[nodiscard]] constexpr T rotl(T x, int s) noexcept;
```

**Constraints:** \( T \) is an unsigned integer type (6.8.2).

Let \( r \) be \( s \% N \).

**Returns:** If \( r \) is 0, \( x \); if \( r \) is positive, \((x << r) | (x >> (N - r))\); if \( r \) is negative, \( \text{rotr}(x, -r) \).

```cpp
template<class T>
[[nodiscard]] constexpr T rotr(T x, int s) noexcept;
```

**Constraints:** \( T \) is an unsigned integer type (6.8.2).

Let \( r \) be \( s \% N \).

**Returns:** If \( r \) is 0, \( x \); if \( r \) is positive, \((x >> r) | (x << (N - r))\); if \( r \) is negative, \( \text{rotl}(x, -r) \).

### 26.5.6 Counting

In the following descriptions, let \( N \) denote `numeric_limits<T>::digits`.

```cpp
template<class T>
 constexpr int countl_zero(T x) noexcept;
```

**Constraints:** \( T \) is an unsigned integer type (6.8.2).

**Returns:** The number of consecutive 0 bits in the value of \( x \), starting from the most significant bit.

**Note 1:** Returns \( N \) if \( x == 0 \). — end note

```cpp
template<class T>
 constexpr int countl_one(T x) noexcept;
```

**Constraints:** \( T \) is an unsigned integer type (6.8.2).

**Returns:** The number of consecutive 1 bits in the value of \( x \), starting from the most significant bit.

**Note 2:** Returns \( N \) if \( x == \text{numeric_limits<T>::max()} \). — end note

```cpp
template<class T>
 constexpr int countr_zero(T x) noexcept;
```

**Constraints:** \( T \) is an unsigned integer type (6.8.2).
Returns: The number of consecutive 0 bits in the value of x, starting from the least significant bit.

[Note 3: Returns N if x == 0. — end note]

template<class T>
constexpr int countr_one(T x) noexcept;

Constraints: T is an unsigned integer type (6.8.2).

Returns: The number of consecutive 1 bits in the value of x, starting from the least significant bit.

[Note 4: Returns N if x == numeric_limits<T>::max(). — end note]

template<class T>
constexpr int popcount(T x) noexcept;

Constraints: T is an unsigned integer type (6.8.2).

Returns: The number of 1 bits in the value of x.

26.5.7 Endian

Two common methods of byte ordering in multibyte scalar types are big-endian and little-endian in the execution environment. Big-endian is a format for storage of binary data in which the most significant byte is placed first, with the rest in descending order. Little-endian is a format for storage of binary data in which the least significant byte is placed first, with the rest in ascending order. This subclause describes the endianness of the scalar types of the execution environment.

enum class endian {
    little = see below,
    big = see below,
    native = see below
};

If all scalar types have size 1 byte, then all of endian::little, endian::big, and endian::native have the same value. Otherwise, endian::little is not equal to endian::big. If all scalar types are big-endian, endian::native is equal to endian::big. If all scalar types are little-endian, endian::native is equal to endian::little. Otherwise, endian::native is not equal to either endian::big or endian::little.

26.6 Random number generation

26.6.1 General

Subclause 26.6 defines a facility for generating (pseudo-)random numbers.

In addition to a few utilities, four categories of entities are described: uniform random bit generators, random number engines, random number engine adaptors, and random number distributions. These categorizations are applicable to types that meet the corresponding requirements, to objects instantiated from such types, and to templates producing such types when instantiated.

[Note 1: These entities are specified in such a way as to permit the binding of any uniform random bit generator object e as the argument to any random number distribution object d, thus producing a zero-argument function object such as given by bind(d,e). — end note]

Each of the entities specified in 26.6 has an associated arithmetic type (6.8.2) identified as result_type. With T as the result_type thus associated with such an entity, that entity is characterized:

(3.1) as boolean or equivalently as boolean-valued, if T is bool;

(3.2) otherwise as integral or equivalently as integer-valued, if numeric_limits<T>::is_integer is true;

(3.3) otherwise as floating-point or equivalently as real-valued.

If integer-valued, an entity may optionally be further characterized as signed or unsigned, according to numeric_limits<T>::is_signed.

Unless otherwise specified, all descriptions of calculations in 26.6 use mathematical real numbers.

Throughout 26.6, the operators bitand, bitor, and xor denote the respective conventional bitwise operations. Further:

(5.1) the operator rshift denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and
(5.2) the operator lshift\(_w\) denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo \(2^w\).

### 26.6.2 Header <random> synopsis

```cpp
#include <initializer_list>

namespace std {

// 26.6.3.3, uniform random bit generator requirements
template<class G>
concept uniform_random_bit_generator = see below;

// 26.6.4.2, class template linear_congruential_engine
template<class UIntType, UIntType a, UIntType c, UIntType m>
class linear_congruential_engine;

// 26.6.4.3, class template mersenne_twister_engine
template<class UIntType, size_t w, size_t n, size_t m, size_t r,
     UIntType a, size_t u, UIntType d, size_t s,
     UIntType b, size_t t,
     UIntType c, size_t l, UIntType f>
class mersenne_twister_engine;

// 26.6.4.4, class template subtract_with_carry_engine
template<class UIntType, size_t w, size_t s, size_t r>
class subtract_with_carry_engine;

// 26.6.5.2, class template discard_block_engine
template<class Engine, size_t p, size_t r>
class discard_block_engine;

// 26.6.5.3, class template independent_bits_engine
template<class Engine, size_t w, class UIntType>
class independent_bits_engine;

// 26.6.5.4, class template shuffle_order_engine
template<class Engine, size_t k>
class shuffle_order_engine;

// 26.6.6, engines and engine adaptors with predefined parameters
using minstd_rand0 = see below;
using minstd_rand = see below;
using mt19937 = see below;
using mt19937_64 = see below;
using ranlux48_base = see below;
using ranlux48_base = see below;
using ranlux24 = see below;
using ranlux48 = see below;
using knuth_b = see below;

using default_random_engine = see below;

// 26.6.7, class random_device
class random_device;

// 26.6.8.1, class seed_seq
class seed_seq;

// 26.6.8.2, function template generate_canonical
template<class RealType, size_t bits, class URBG>
RealType generate_canonical(URBG& g);
```
// 26.6.9.2.1, class template uniform_int_distribution
template<class IntType = int>
  class uniform_int_distribution;

// 26.6.9.2.2, class template uniform_real_distribution
template<class RealType = double>
  class uniform_real_distribution;

// 26.6.9.3.1, class bernoulli_distribution
class bernoulli_distribution;

// 26.6.9.3.2, class template binomial_distribution
template<class IntType = int>
  class binomial_distribution;

// 26.6.9.3.3, class template geometric_distribution
template<class IntType = int>
  class geometric_distribution;

// 26.6.9.3.4, class template negative_binomial_distribution
template<class IntType = int>
  class negative_binomial_distribution;

// 26.6.9.4.1, class template poisson_distribution
template<class IntType = int>
  class poisson_distribution;

// 26.6.9.4.2, class template exponential_distribution
template<class RealType = double>
  class exponential_distribution;

// 26.6.9.4.3, class template gamma_distribution
template<class RealType = double>
  class gamma_distribution;

// 26.6.9.4.4, class template weibull_distribution
template<class RealType = double>
  class weibull_distribution;

// 26.6.9.4.5, class template extreme_value_distribution
template<class RealType = double>
  class extreme_value_distribution;

// 26.6.9.5.1, class template normal_distribution
template<class RealType = double>
  class normal_distribution;

// 26.6.9.5.2, class template lognormal_distribution
template<class RealType = double>
  class lognormal_distribution;

// 26.6.9.5.3, class template chi_squared_distribution
template<class RealType = double>
  class chi_squared_distribution;

// 26.6.9.5.4, class template cauchy_distribution
template<class RealType = double>
  class cauchy_distribution;

// 26.6.9.5.5, class template fisher_f_distribution
template<class RealType = double>
  class fisher_f_distribution;
26.6.3 Requirements [rand.req]

26.6.3.1 General requirements [rand.req.genl]

1 Throughout this subclause 26.6, the effect of instantiating a template:

1.1 that has a template type parameter named Sseq is undefined unless the corresponding template argument is cv-unqualified and meets the requirements of seed sequence (26.6.3.2).

1.2 that has a template type parameter named URBG is undefined unless the corresponding template argument is cv-unqualified and meets the requirements of uniform random bit generator (26.6.3.3).

1.3 that has a template type parameter named Engine is undefined unless the corresponding template argument is cv-unqualified and meets the requirements of random number engine (26.6.3.4).

1.4 that has a template type parameter named RealType is undefined unless the corresponding template argument is cv-unqualified and is one of float, double, or long double.

1.5 that has a template type parameter named IntType is undefined unless the corresponding template argument is cv-unqualified and is one of short, int, long, long long, unsigned short, unsigned int, unsigned long, or unsigned long long.

1.6 that has a template type parameter named UIntType is undefined unless the corresponding template argument is cv-unqualified and is one of unsigned short, unsigned int, unsigned long, or unsigned long long.

2 Throughout this subclause 26.6, phrases of the form “x is an iterator of a specific kind” shall be interpreted as equivalent to the more formal requirement that “x is a value of a type meeting the requirements of the specified iterator type”.

3 Throughout this subclause 26.6, any constructor that can be called with a single argument and that meets a requirement specified in this subclause shall be declared explicit.

26.6.3.2 Seed sequence requirements [rand.req.seedseq]

1 A seed sequence is an object that consumes a sequence of integer-valued data and produces a requested number of unsigned integer values \( i, 0 \leq i < 2^{32} \), based on the consumed data.

[Note 1: Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful, for example, in applications requiring large numbers of random number engines. – end note]

2 A class S meets the requirements of a seed sequence if the expressions shown in Table 93 are valid and have the indicated semantics, and if S also meets all other requirements of this subclause 26.6.3.2. In that Table and throughout this subclause:

2.1 T is the type named by S’s associated result_type;

2.2 q is a value of S and r is a possibly const value of S;

2.3 ib and ie are input iterators with an unsigned integer value_type of at least 32 bits;

2.4 rb and re are mutable random access iterators with an unsigned integer value_type of at least 32 bits;

2.5 ob is an output iterator; and

2.6 il is a value of initializer_list<T>.
Table 93: Seed sequence requirements  [tab:rand.req.seedseq]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>S::result_type</td>
<td>T</td>
<td>T is an unsigned integer type (6.8.2) of at least 32 bits.</td>
<td>compile-time</td>
</tr>
<tr>
<td>S()</td>
<td></td>
<td>Creates a seed sequence with the same initial state as all other default-constructed seed sequences of type S.</td>
<td>constant</td>
</tr>
<tr>
<td>S(ib,ie)</td>
<td></td>
<td>Creates a seed sequence having internal state that depends on some or all of the bits of the supplied sequence [ib,ie).</td>
<td>(O(ie - ib))</td>
</tr>
<tr>
<td>S(il)</td>
<td></td>
<td>Same as (S(il.begin(), il.end())). Same as (S(il.begin(), il.end())).</td>
<td></td>
</tr>
<tr>
<td>q.generate(rb,re)</td>
<td>void</td>
<td>Does nothing if (rb == re). Otherwise, fills the supplied sequence ([rb,re]) with 32-bit quantities that depend on the sequence supplied to the constructor and possibly also depend on the history of generate's previous invocations.</td>
<td>(O(re - rb))</td>
</tr>
<tr>
<td>r.size()</td>
<td>size_t</td>
<td>The number of 32-bit units that would be copied by a call to r.param.</td>
<td>constant</td>
</tr>
<tr>
<td>r.param(ob)</td>
<td>void</td>
<td>Copies to the given destination a sequence of 32-bit units that can be provided to the constructor of a second object of type S, and that would reproduce in that second object a state indistinguishable from the state of the first object.</td>
<td>(O(r.size()))</td>
</tr>
</tbody>
</table>

26.6.3.3 Uniform random bit generator requirements  [rand.req.urng]

1 A uniform random bit generator \(g\) of type \(G\) is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned.

[Note 1: The degree to which \(g\)'s results approximate the ideal is often determined statistically. — end note]

```cpp
template<class G>
concept uniform_random_bit_generator =
    invocable<G&> && unsigned_integral<invoke_result_t<G&>> &&
    requires {
    { G::min() } -> same_as<invoke_result_t<G&>>;
    { G::max() } -> same_as<invoke_result_t<G&>>;
    requires bool_constant<(G::min() < G::max())>::value;
    };
```

2 Let \(g\) be an object of type \(G\). \(G\) models uniform_random_bit_generator only if

(2.1) \(G::min() \leq g()\),
(2.2) \(g() \leq G::max()\), and
(2.3) \(g()\) has amortized constant complexity.

3 A class \(G\) meets the uniform random bit generator requirements if \(G\) models uniform_random_bit_generator, invoke_result_t<G&> is an unsigned integer type (6.8.2), and \(G\) provides a nested typedef-name result_type that denotes the same type as invoke_result_t<G&>. 

§ 26.6.3.3
26.6.3.4 Random number engine requirements

A random number engine (commonly shortened to engine) \(e\) of type \(E\) is a uniform random bit generator that additionally meets the requirements (e.g., for seeding and for input/output) specified in this subclause.

At any given time, \(e\) has a state \(e_i\) for some integer \(i \geq 0\). Upon construction, \(e\) has an initial state \(e_0\). An engine’s state may be established via a constructor, a seed function, assignment, or a suitable operator\(>>\).

E’s specification shall define:

1. the size of \(E\)’s state in multiples of the size of result_type, given as an integral constant expression;
2. the transition algorithm \(TA\) by which \(e\)’s state \(e_i\) is advanced to its successor state \(e_{i+1}\); and
3. the generation algorithm \(GA\) by which an engine’s state is mapped to a value of type result_type.

A class \(E\) that meets the requirements of a uniform random bit generator (26.6.3.3) also meets the requirements of a random number engine if the expressions shown in Table 94 are valid and have the indicated semantics, and if \(E\) also meets all other requirements of this subclause 26.6.3.4. In that Table and throughout this subclause:

- \(T\) is the type named by \(E\)’s associated result_type;
- \(e\) is a value of \(E\), \(v\) is an lvalue of \(E\), \(x\) and \(y\) are (possibly const) values of \(E\);
- \(s\) is a value of \(T\);
- \(q\) is an lvalue meeting the requirements of a seed sequence (26.6.3.2);
- \(z\) is a value of type unsigned long long;
- \(os\) is an lvalue of the type of some class template specialization basic_ostream<charT, traits>; and
- \(is\) is an lvalue of the type of some class template specialization basic_istream<charT, traits>;

where charT and traits are constrained according to Clause 21 and Clause 29.

Table 94: Random number engine requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E())</td>
<td></td>
<td>Creates an engine with the same initial state as all other default-constructed engines of type (E).</td>
<td>(O(\text{size of state}))</td>
</tr>
<tr>
<td>(E(x))</td>
<td></td>
<td>Creates an engine that compares equal to (x).</td>
<td>(O(\text{size of state}))</td>
</tr>
<tr>
<td>(E(s))</td>
<td></td>
<td>Creates an engine with initial state determined by (s).</td>
<td>(O(\text{size of state}))</td>
</tr>
<tr>
<td>(E(q))</td>
<td></td>
<td>Creates an engine with an initial state that depends on a sequence produced by one call to (q).generate.</td>
<td>same as complexity of (q).generate called on a sequence whose length is size of state</td>
</tr>
<tr>
<td>(e().seed())</td>
<td>void</td>
<td>Postconditions: (e == E()).</td>
<td>same as (E())</td>
</tr>
<tr>
<td>(e().seed(s))</td>
<td>void</td>
<td>Postconditions: (e == E(s)).</td>
<td>same as (E(s))</td>
</tr>
<tr>
<td>(e().seed(q))</td>
<td>void</td>
<td>Postconditions: (e == E(q)).</td>
<td>same as (E(q))</td>
</tr>
<tr>
<td>(e())</td>
<td>(T)</td>
<td>Advances (e)’s state (e_i) to (e_{i+1}) = (TA(e_i)) and returns (GA(e_i)).</td>
<td>per 26.6.3.3</td>
</tr>
<tr>
<td>(e().discard(z))</td>
<td>void</td>
<td>Advances (e)’s state (e_i) to (e_{i+z}) by any means equivalent to (z) consecutive calls (e()).</td>
<td>no worse than the complexity of (z) consecutive calls (e())</td>
</tr>
</tbody>
</table>

247 This constructor (as well as the subsequent corresponding seed() function) can be particularly useful to applications requiring a large number of independent random sequences.

248 This operation is common in user code, and can often be implemented in an engine-specific manner so as to provide significant performance improvements over an equivalent naive loop that makes \(z\) consecutive calls \(e()\).
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>x == y</td>
<td>bool</td>
<td>This operator is an equivalence relation. With ( S_x ) and ( S_y ) as the infinite sequences of values that would be generated by repeated future calls to ( x() ) and ( y() ), respectively, returns true if ( S_x = S_y ); else returns false.</td>
<td>( \Theta ) (size of state)</td>
</tr>
<tr>
<td>x != y</td>
<td>bool</td>
<td>! (x == y)</td>
<td>( \Theta ) (size of state)</td>
</tr>
<tr>
<td>os &lt;&lt; x</td>
<td>reference to the type of os</td>
<td>With ( \text{os.fmtflags} ) set to ( \text{ios_base::dec</td>
<td>ios_base::left} ) and the fill character set to the space character, writes to ( \text{os} ) the textual representation of ( x )'s current state. In the output, adjacent numbers are separated by one or more space characters. Postconditions: The ( \text{os.fmtflags} ) and fill character are unchanged.</td>
</tr>
<tr>
<td>is &gt;&gt; v</td>
<td>reference to the type of is</td>
<td>With ( \text{is.fmtflags} ) set to ( \text{ios_base::dec} ), sets ( v )'s state as determined by reading its textual representation from ( \text{is} ). If bad input is encountered, ensures that ( v )'s state is unchanged by the operation and calls ( \text{is.setstate(ios_base::failbit)} ) (which may throw ( \text{ios_base::failure} ) (29.5.5.4)). If a textual representation written via ( \text{os} ) was subsequently read via ( \text{is} ), then ( x == v ) provided that there have been no intervening invocations of ( x ) or of ( v ). Preconditions: ( \text{is} ) provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of ( \text{is} ), and whose type's template specialization arguments ( \text{charT} ) and ( \text{traits} ) were respectively the same as those of ( \text{is} ). Postconditions: The ( \text{is.fmtflags} ) are unchanged.</td>
<td>( \Theta ) (size of state)</td>
</tr>
</tbody>
</table>

5 E shall meet the Cpp17CopyConstructible (Table 29) and Cpp17CopyAssignible (Table 31) requirements. These operations shall each be of complexity no worse than \( \Theta \) (size of state).

26.6.3.5 Random number engine adaptor requirements [rand.req.adapt]

1 A random number engine adaptor (commonly shortened to adaptor) \( a \) of type \( A \) is a random number engine that takes values produced by some other random number engine, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. An engine \( b \) of type \( B \) adapted in this way is termed a base engine in this context. The expression \( a.based() \) shall be valid and shall return a const reference to \( a \)'s base engine.
The requirements of a random number engine type shall be interpreted as follows with respect to a random number engine adaptor type.

```cpp
A::A();
```

**Effects:** The base engine is initialized as if by its default constructor.

```cpp
bool operator==(const A& a1, const A& a2);
```

**Returns:** `true` if `a1`'s base engine is equal to `a2`'s base engine. Otherwise returns `false`.

```cpp
A::A(result_type s);
```

**Effects:** The base engine is initialized with `s`.

```cpp
template<class Sseq> A::A(Sseq& q);
```

**Effects:** The base engine is initialized with `q`.

```cpp
void seed();
```

**Effects:** With `b` as the base engine, invokes `b.seed()`.

```cpp
void seed(result_type s);
```

**Effects:** With `b` as the base engine, invokes `b.seed(s)`.

```cpp
template<class Sseq> void seed(Sseq& q);
```

**Effects:** With `b` as the base engine, invokes `b.seed(q)`.

A shall also meet the following additional requirements:

- The complexity of each function shall not exceed the complexity of the corresponding function applied to the base engine.
- The state of `A` shall include the state of its base engine. The size of `A`'s state shall be no less than the size of the base engine.
- Copying `A`'s state (e.g., during copy construction or copy assignment) shall include copying the state of the base engine of `A`.
- The textual representation of `A` shall include the textual representation of its base engine.

### 26.6.3.6 Random number distribution requirements

A random number distribution (commonly shortened to distribution) `d` of type `D` is a function object returning values that are distributed according to an associated mathematical probability density function `p(z)` or according to an associated discrete probability function `P(z_i)`. A distribution’s specification identifies its associated probability function `p(z)` or `P(z_i)`.

An associated probability function is typically expressed using certain externally-supplied quantities known as the parameters of the distribution. Such distribution parameters are identified in this context by writing, for example, `p(z | a, b)` or `P(z_i | a, b)`, to name specific parameters, or by writing, for example, `p(z | {p})` or `P(z_i | {p})`, to denote a distribution’s parameters `p` taken as a whole.

A class `D` meets the requirements of a random number distribution if the expressions shown in Table 95 are valid and have the indicated semantics, and if `D` and its associated types also meet all other requirements of this subclause 26.6.3.6. In that Table and throughout this subclause,

- `T` is the type named by `D`'s associated `result_type`;
- `P` is the type named by `D`’s associated `param_type`;
- `d` is a value of `D`, and `x` and `y` are (possibly `const`) values of `D`;
- `glb` and `lub` are values of `T` respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by `d`’s `operator()`, as determined by the current values of `d`’s parameters;
- `p` is a (possibly `const`) value of `P`;
- `g`, `g1`, and `g2` are lvalues of a type meeting the requirements of a uniform random bit generator (26.6.3.3);
- `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and
— *is* is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;
where `charT` and `traits` are constrained according to Clause 21 and Clause 29.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>D::result_type</code></td>
<td>T</td>
<td>T is an arithmetic type (6.8.2).</td>
<td>compile-time</td>
</tr>
<tr>
<td><code>D::param_type</code></td>
<td>P</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td><code>D()</code></td>
<td></td>
<td>Creates a distribution whose behavior is indistinguishable from that of any other newly default-constructed distribution of type <code>D</code>.</td>
<td>constant</td>
</tr>
<tr>
<td><code>D(p)</code></td>
<td></td>
<td>Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct <code>p</code>.</td>
<td>same as <code>p</code>'s construction</td>
</tr>
<tr>
<td><code>d.reset()</code></td>
<td>void</td>
<td>Subsequent uses of <code>d</code> do not depend on values produced by any engine prior to invoking <code>reset</code>.</td>
<td>constant</td>
</tr>
<tr>
<td><code>x.param()</code></td>
<td>P</td>
<td>Returns a value <code>p</code> such that <code>D(p).param() == p</code>.</td>
<td>no worse than the complexity of <code>D(p)</code></td>
</tr>
<tr>
<td><code>d.param(p)</code></td>
<td>void</td>
<td>*Postconditions: <code>d.param() == p</code>.</td>
<td>no worse than the complexity of <code>D(p)</code></td>
</tr>
<tr>
<td><code>d(g)</code></td>
<td>T</td>
<td>With <code>p = d.param()</code>, the sequence of numbers returned by successive invocations with the same object <code>g</code> is randomly distributed according to the associated `p(z</td>
<td>{p})<code>or</code>P(z_i</td>
</tr>
<tr>
<td><code>d(g,p)</code></td>
<td>T</td>
<td>The sequence of numbers returned by successive invocations with the same objects <code>g</code> and <code>p</code> is randomly distributed according to the associated `p(z</td>
<td>{p})<code>or</code>P(z_i</td>
</tr>
<tr>
<td><code>x.min()</code></td>
<td>T</td>
<td>Returns <code>glob</code>.</td>
<td>constant</td>
</tr>
<tr>
<td><code>x.max()</code></td>
<td>T</td>
<td>Returns <code>lub</code>.</td>
<td>constant</td>
</tr>
<tr>
<td><code>x == y</code></td>
<td>bool</td>
<td>This operator is an equivalence relation. Returns <code>true</code> if <code>x.param() == y.param()</code> and <code>S_1 = S_2</code>, where <code>S_1</code> and <code>S_2</code> are the infinite sequences of values that would be generated, respectively, by repeated future calls to <code>x(g1)</code> and <code>y(g2)</code> whenever <code>g1 == g2</code>. Otherwise returns <code>false</code>.</td>
<td>constant</td>
</tr>
<tr>
<td><code>x != y</code></td>
<td>bool</td>
<td>!(x == y).</td>
<td>same as <code>x == y</code></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Pre/post-condition</td>
<td>Complexity</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>--------------------</td>
<td>------------</td>
</tr>
<tr>
<td><code>os &lt;&lt; x</code></td>
<td>reference to the type of <code>os</code></td>
<td>Writes to <code>os</code> a textual representation for the parameters and the additional internal data of <code>x</code>. Postconditions: The <code>os.</code>fmtflags and fill character are unchanged.</td>
<td></td>
</tr>
<tr>
<td><code>is &gt;&gt; d</code></td>
<td>reference to the type of <code>is</code></td>
<td>Restores from <code>is</code> the parameters and additional internal data of the lvalue <code>d</code>. If bad input is encountered, ensures that <code>d</code> is unchanged by the operation and calls <code>is.setstate(ios_base::failbit)</code> (which may throw <code>ios_base::failure (29.5.5.4)</code>). Preconditions: <code>is</code> provides a textual representation that was previously written using an <code>os</code> whose imbued locale and whose type’s template specialization arguments charT and traits were the same as those of <code>is</code>. Postconditions: The <code>is.</code>fmtflags are unchanged.</td>
<td></td>
</tr>
</tbody>
</table>

4 D shall meet the *Cpp17CopyConstructible* (Table 29) and *Cpp17CopyAssignable* (Table 31) requirements.

5 The sequence of numbers produced by repeated invocations of `d(g)` shall be independent of any invocation of `os << d` or of any const member function of `D` between any of the invocations `d(g)`.

6 If a textual representation is written using `os << x` and that representation is restored into the same or a different object `y` of the same type using `is >> y`, repeated invocations of `y(g)` shall produce the same sequence of numbers as would repeated invocations of `x(g)`.

7 It is unspecified whether `D::param_type` is declared as a (nested) class or via a typedef. In this subclause 26.6, declarations of `D::param_type` are in the form of typedefs for convenience of exposition only.

8 P shall meet the *Cpp17CopyConstructible* (Table 29), *Cpp17CopyAssignable* (Table 31), and *Cpp17Equality-Comparable* (Table 25) requirements.

9 For each of the constructors of `D` taking arguments corresponding to parameters of the distribution, P shall have a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values. Moreover, for each of the member functions of `D` that return values corresponding to parameters of the distribution, P shall have a corresponding member function with the identical name, type, and semantics.

10 P shall have a declaration of the form

```
using distribution_type = D;
```

#### 26.6.4 Random number engine class templates

##### 26.6.4.1 General

1 Each type instantiated from a class template specified in 26.6.4 meets the requirements of a random number engine (26.6.3.4) type.

2 Except where specified otherwise, the complexity of each function specified in 26.6.4 is constant.

3 Except where specified otherwise, no function described in 26.6.4 throws an exception.

4 Every function described in 26.6.4 that has a function parameter `q` of type `seq` for a template type parameter named `seq` that is different from type `seed_seq` throws what and when the invocation of `q.generate` throws.
Descriptions are provided in 26.6.4 only for engine operations that are not described in 26.6.3.4 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

Each template specified in 26.6.4 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

For every random number engine and for every random number engine adaptor X defined in 26.6.4 and in 26.6.5:

(7.1) — if the constructor

```cpp
template<class Sseq> explicit X(Sseq& q);
```

is called with a type Sseq that does not qualify as a seed sequence, then this constructor shall not participate in overload resolution;

(7.2) — if the member function

```cpp
template<class Sseq> void seed(Sseq& q);
```

is called with a type Sseq that does not qualify as a seed sequence, then this function shall not participate in overload resolution.

The extent to which an implementation determines that a type cannot be a seed sequence is unspecified, except that as a minimum a type shall not qualify as a seed sequence if it is implicitly convertible to X::result_type.

26.6.4.2 Class template linear_congruential_engine

A linear_congruential_engine random number engine produces unsigned integer random numbers. The state \(x_i\) of a linear_congruential_engine object \(x\) is of size 1 and consists of a single integer. The transition algorithm is a modular linear function of the form \(TA(x_i) = (a \cdot x_i + c) \mod m\); the generation algorithm is \(GA(x_i) = x_i + 1\).

```cpp
template<class UIntType, UIntType a, UIntType c, UIntType m>
class linear_congruential_engine {
public:
  // types
  using result_type = UIntType;

  // engine characteristics
  static constexpr result_type multiplier = a;
  static constexpr result_type increment = c;
  static constexpr result_type modulus = m;
  static constexpr result_type min() { return c == 0u ? 1u: 0u; }
  static constexpr result_type max() { return m - 1u; }
  static constexpr result_type default_seed = 1u;

  // constructors and seeding functions
  linear_congruential_engine() : linear_congruential_engine(default_seed) {}
  explicit linear_congruential_engine(result_type s);
  template<class Sseq> explicit linear_congruential_engine(Sseq& q);
  void seed(result_type s = default_seed);
  template<class Sseq> void seed(Sseq& q);

  // generating functions
  result_type operator()();
  void discard(unsigned long long z);
};
```

If the template parameter \(m\) is 0, the modulus \(m\) used throughout this subclause 26.6.4.2 is `numeric_limits<result_type>::max()` plus 1.

[Note 1: \(m\) need not be representable as a value of type `result_type`. — end note]

If the template parameter \(m\) is not 0, the following relations shall hold: \(a < m\) and \(c < m\).

The textual representation consists of the value of \(x_i\).
explicit linear_congruential_engine(result_type s);

Effects: If \( c \mod m = 0 \) and \( s \mod m = 0 \), sets the engine's state to 1, otherwise sets the engine's state to \( s \mod m \).

template<class Sseq> explicit linear_congruential_engine(Sseq& q);

Effects: With \( k = \left\lfloor \frac{\log m}{32} \right\rfloor \) and \( a \) an array (or equivalent) of length \( k + 3 \), invokes \( q \).generate(\( a + 0 \), \( a + k + 3 \)) and then computes \( S = \left( \sum_{j=0}^{k-1} a_{j+3} \cdot 2^{24} \right) \mod m \). If \( c \mod m = 0 \) and \( S = 0 \), sets the engine's state to 1, else sets the engine's state to \( S \).

### 26.6.4.3 Class template mersenne_twister_engine

A `mersenne_twister_engine` random number engine\(^{249}\) produces unsigned integer random numbers in the closed interval \([0, 2^w - 1]\). The state \( x_0 \) of a `mersenne_twister_engine` object \( x \) is of size \( n \) and consists of a sequence \( X \) of \( n \) values of the type delivered by \( x \); all subscripts applied to \( X \) are to be taken modulo \( n \).

The transition algorithm employs a twisted generalized feedback shift register defined by shift values \( n \) and \( m \), a twist value \( \ell \), and a conditional xor-mask \( a \). To improve the uniformity of the result, the bits of the raw shift register are additionally tempered (i.e., scrambled) according to a bit-scrambling matrix defined by values \( u \), \( d \), \( s \), \( b \), \( c \), and \( \ell \).

The state transition is performed as follows:

### 26.6.4.3.1 Class template mersenne_twister_engine

1. The state transition is performed as follows:

   \[
   X_i = X_i \cdot (Y \text{ bitand } 1), \quad \text{for } i = 0, 1, \ldots, n-1
   \]

2. The state transition is performed as follows:

   \[
   X_i = X_i \cdot (Y \text{ bitand } 1), \quad \text{for } i = 0, 1, \ldots, n-1
   \]

3. The state transition is performed as follows:

   \[
   X_i = X_i \cdot (Y \text{ bitand } 1), \quad \text{for } i = 0, 1, \ldots, n-1
   \]

4. The state transition is performed as follows:

   \[
   X_i = X_i \cdot (Y \text{ bitand } 1), \quad \text{for } i = 0, 1, \ldots, n-1
   \]

5. The state transition is performed as follows:

   \[
   X_i = X_i \cdot (Y \text{ bitand } 1), \quad \text{for } i = 0, 1, \ldots, n-1
   \]

6. The state transition is performed as follows:

   \[
   X_i = X_i \cdot (Y \text{ bitand } 1), \quad \text{for } i = 0, 1, \ldots, n-1
   \]

\(^{249}\) The name of this engine refers, in part, to a property of its period: For properly-selected values of the parameters, the period is closely related to a large Mersenne prime number.
The generation algorithm is given by

\[
\begin{align*}
X_{i+1} &= (a \cdot X_i + b) \mod 2^w, \\
Y &= (X_{i+1} \mod m) / m, \\
\end{align*}
\]

where \(a\), \(b\), \(m\), and \(w\) are constants. The state transition is performed as follows:

1. **Let** \( Y = X_{i-s} - X_{i-r} - c \).
2. **Set** \( X_i \) to \( y = Y \mod m \). **Set** \( c \) to 1 if \( Y < 0 \), otherwise **set** \( c \) to 0.

Contents:

- **Section 26.6.4.4** Class template `subtract_with_carry_engine`

The state \( x_i \) of a `subtract_with_carry_engine` object \( x \) is of size \( \Theta(r) \), and consists of a sequence \( X \) of \( r \) integer values \( 0 \leq X_i < m = 2^w \); all subscripts applied to \( X \) are to be taken modulo \( r \). The state \( x_i \) additionally consists of an integer \( c \) (known as the `carry`) whose value is either 0 or 1.

The state transition is performed as follows:

1. **Let** \( Y = X_{i-s} - X_{i-r} - c \).
2. **Set** \( X_i \) to \( y = Y \mod m \). **Set** \( c \) to 1 if \( Y < 0 \), otherwise **set** \( c \) to 0.

**Note 1:** This algorithm corresponds to a modular linear function of the form \( TA(x_i) = (a \cdot x_i) \mod b \), where \( b \) is of the form \( m' - m^* + 1 \) and \( a = b \mod (b - 1)/m' \). 

The generation algorithm is given by \( GA(x_i) = y \), where \( y \) is the value produced as a result of advancing the engine's state as described above.

```cpp
template<class UIntType, size_t r, size_t s, size_t t>
class subtract_with_carry_engine {
public:
    // types
    using result_type = UIntType;

    // engine characteristics
    static constexpr size_t word_size = t;
    static constexpr size_t short_lag = s;
    static constexpr size_t long_lag = r;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return m - 1; }
    static constexpr result_type default_seed = 19780530u;
};
```
// constructors and seeding functions
subtract_with_carry_engine() : subtract_with_carry_engine(default_seed) {}
explicit subtract_with_carry_engine(result_type value);
template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);
void seed(result_type value = default_seed);
template<class Sseq> void seed(Sseq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);

The following relations shall hold: 0u < s, s < r, 0 < w, and w <= numeric_limits<UIntType>::digits.
The textual representation consists of the values of $X_{i-r},\ldots,X_{i-1}$, in that order, followed by $c$.

explicit subtract_with_carry_engine(result_type value);

Effects: Sets the values of $X_{i-r},\ldots,X_{i-1}$, in that order, as specified below. If $X_{i-1}$ is then 0, sets $c$ to 1; otherwise sets $c$ to 0.

To set the values $X_k$, first construct $e$, a linear_congruential_engine object, as if by the following definition:

```
linear_congruential_engine<result_type,
40014u,0u,2147483563u> e(value == 0u ? default_seed : value);
```

Then, to set each $X_k$, obtain new values $z_0,\ldots,z_{n-1}$ from $n = \lceil w/32 \rceil$ successive invocations of $e$ taken modulo $2^{32}$. Set $X_k$ to $\left(\sum_{j=0}^{n-1} z_j \cdot 2^{32j}\right) \mod m$.

Complexity: Exactly $n \cdot r$ invocations of $e$.

template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);

Effects: With $k = \lceil w/32 \rceil$ and $a$ an array (or equivalent) of length $r \cdot k$, invokes $q\_generate(a+0, a+r \cdot k)$ and then, iteratively for $i = r,\ldots,-1$, sets $X_i$ to $\left(\sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32j}\right) \mod m$. If $X_{i-1}$ is then 0, sets $c$ to 1; otherwise sets $c$ to 0.

26.6.5 Random number engine adaptor class templates [rand.adapt]

26.6.5.1 In general [rand.adapt.general]

1 Each type instantiated from a class template specified in this subclause 26.6.5 meets the requirements of a random number engine adaptor (26.6.3.5) type.

2 Except where specified otherwise, the complexity of each function specified in this subclause 26.6.5 is constant.

3 Except where specified otherwise, no function described in this subclause 26.6.5 throws an exception.

4 Every function described in this subclause 26.6.5 that has a function parameter $q$ of type $Sseq$ for a template type parameter named $Sseq$ that is different from type $seed\_seq$ throws what and when the invocation of $q\_generate$ throws.

5 Descriptions are provided in this subclause 26.6.5 only for adaptor operations that are not described in subclause 26.6.3.5 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

6 Each template specified in this subclause 26.6.5 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

26.6.5.2 Class template discard_block_engine [rand.adapt.disc]

A discard_block_engine random number engine adaptor produces random numbers selected from those produced by some base engine $e$. The state $x_i$ of a discard_block_engine engine adaptor object $x$ consists of the state $e_i$ of its base engine $e$ and an additional integer $n$. The size of the state is the size of $e$’s state plus 1.
The transition algorithm discards all but \( r > 0 \) values from each block of \( p \geq r \) values delivered by \( e \). The state transition is performed as follows: If \( n \geq r \), advance the state of \( e \) from \( e_i \) to \( e_{i+p-r} \) and set \( n \) to 0. In any case, then increment \( n \) and advance \( e \)'s then-current state \( e_j \) to \( e_{j+1} \).

The generation algorithm yields the value returned by the last invocation of \( e() \) while advancing \( e \)'s state as described above.

```cpp
template<class Engine, size_t p, size_t r>
class discard_block_engine {
public:
    // types
    using result_type = typename Engine::result_type;

    // engine characteristics
    static constexpr size_t block_size = p;
    static constexpr size_t used_block = r;
    static constexpr result_type min() { return Engine::min(); }
    static constexpr result_type max() { return Engine::max(); }

    // constructors and seeding functions
    discard_block_engine();
    explicit discard_block_engine(const Engine& e);
    explicit discard_block_engine(Engine&& e);
    explicit discard_block_engine(result_type s);
    template<class Sseq> explicit discard_block_engine(Sseq& q);
    void seed();
    void seed(result_type s);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const Engine& base() const noexcept { return e; }

private:
    Engine e;  // exposition only
    int n;  // exposition only
};
```

The following relations shall hold: \( 0 < r \) and \( r \leq p \).

The textual representation consists of the textual representation of \( e \) followed by the value of \( n \).

In addition to its behavior pursuant to subclause 26.6.3.5, each constructor that is not a copy constructor sets \( n \) to 0.

### 26.6.5.3 Class template independent_bits_engine

An `independent_bits_engine` random number engine adaptor combines random numbers that are produced by some base engine \( e \), so as to produce random numbers with a specified number of bits \( w \). The state \( x_i \) of an `independent_bits_engine` engine adaptor object \( x \) consists of the state \( e_i \) of its base engine \( e \); the size of the state is the size of \( e \)'s state.

The transition and generation algorithms are described in terms of the following integral constants:

\[ R = e\text{.max()} - e\text{.min()} + 1 \]
\[ m = \lceil \log_2 R \rceil. \]
\[ n_0 = n - w \mod n, \]
\[ y_0 = 2^{w_0} \left\lfloor \frac{R}{2^{w_0}} \right\rfloor, \]
\[ y_1 = 2^{w_0+1} \left\lfloor \frac{R}{2^{w_0+1}} \right\rfloor. \]

Let \( n = \lceil w/m \rceil \) if and only if the relation \( R - y_0 \leq |y_0/n| \) holds as a result. Otherwise let \( n = 1 + \lceil w/m \rceil. \) [Note 1: The relation \( w = n_0 w_0 + (n - n_0)(w_0 + 1) \) always holds. — end note]

The transition algorithm is carried out by invoking \( e() \) as often as needed to obtain \( n_0 \) values less than \( y_0 + e\text{.min()} \) and \( n - n_0 \) values less than \( y_1 + e\text{.min()} \).
The generation algorithm uses the values produced while advancing the state as described above to yield a quantity $S$ obtained as if by the following algorithm:

\[
S = 0;
\]

for \( (k = 0; k \neq n_0; k += 1) \) {
    do $u = e() - e_{\text{min}}();$ while \( (u \geq y_0) \);
    $S = 2^{w_0} \cdot S + u \mod 2^{w_0};$
}

for \( (k = n_0; k \neq n; k += 1) \) {
    do $u = e() - e_{\text{min}}();$ while \( (u \geq y_1) \);
    $S = 2^{w_0+1} \cdot S + u \mod 2^{w_0+1};$
}

```cpp
template<class Engine, size_t w, class UIntType>
class independent_bits_engine {
public:
    // types
    using result_type = UIntType;

    // engine characteristics
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return \(2^w - 1\); }

    // constructors and seeding functions
    independent_bits_engine();
    explicit independent_bits_engine(const Engine& e);
    explicit independent_bits_engine(Engine&& e);
    explicit independent_bits_engine(result_type s);
    template<class Sseq> explicit independent_bits_engine(Sseq& q);
    void seed();
    void seed(result_type s);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const Engine& base() const noexcept { return e; }
private:
    Engine e; // exposition only
};
```

The following relations shall hold: \(0 < w \) and \(w \leq \text{numeric\_limits<result\_type>::digits}\).

The textual representation consists of the textual representation of $e$.

### 26.6.5.4 Class template shuffle_order_engine [rand.adapt.shuf]

A `shuffle_order_engine` random number engine adaptor produces the same random numbers that are produced by some base engine $e$, but delivers them in a different sequence. The state $x_i$ of a `shuffle_order_engine` engine adaptor object $x$ consists of the state $e_i$ of its base engine $e$, an additional value $Y$ of the type delivered by $e$, and an additional sequence $V$ of $k$ values also of the type delivered by $e$. The size of the state is the size of $e$'s state plus $k + 1$.

The transition algorithm permutes the values produced by $e$. The state transition is performed as follows:

1. Calculate an integer $j = \left\lfloor \frac{k(Y - e_{\text{min}})}{e_{\text{max}} - e_{\text{min}} + 1} \right\rfloor$.
2. Set $Y$ to $V_j$ and then set $V_j$ to $e()$.

The generation algorithm yields the last value of $Y$ produced while advancing $e$'s state as described above.
// engine characteristics
static constexpr size_t table_size = k;
static constexpr result_type min() { return Engine::min(); }
static constexpr result_type max() { return Engine::max(); }

// constructors and seeding functions
shuffle_order_engine();
explicit shuffle_order_engine(const Engine& e);
explicit shuffle_order_engine(Engine&& e);
explicit shuffle_order_engine(result_type s);
template<class Sseq> explicit shuffle_order_engine(Sseq& q);
void seed();
void seed(result_type s);
template<class Sseq> void seed(Sseq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);

// property functions
const Engine& base() const noexcept { return e; }

private:
Engine e;            // exposition only
result_type V[k];    // exposition only
result_type Y;       // exposition only

4 The following relation shall hold: $0 < k$.
5 The textual representation consists of the textual representation of $e$, followed by the $k$ values of $V$, followed by the value of $Y$.
6 In addition to its behavior pursuant to subclause 26.6.3.5, each constructor that is not a copy constructor initializes $V[0], \ldots, V[k-1]$ and $Y$, in that order, with values returned by successive invocations of $e()$.

### 26.6.6 Engines and engine adaptors with predefined parameters

using minstd_rand0 =
linear_congruential_engine<uint_fast32_t, 16'807, 0, 2'147'483'647>;

1 Required behavior: The $10000^{th}$ consecutive invocation of a default-constructed object of type `minstd_rand0` produces the value 1043618065.

using minstd_rand =
linear_congruential_engine<uint_fast32_t, 48'271, 0, 2'147'483'647>;

2 Required behavior: The $10000^{th}$ consecutive invocation of a default-constructed object of type `minstd_rand` produces the value 399268537.

using mt19937 =
mersenne_twister_engine<uint_fast32_t, 32, 624, 397, 31,
0x9908'b0df, 11, 0xffffffff, 7, 0x9d2c'5680, 15, 0xfec6'0000, 18, 1'812'433'253>;

3 Required behavior: The $10000^{th}$ consecutive invocation of a default-constructed object of type `mt19937` produces the value 4123659995.

using mt19937_64 =
mersenne_twister_engine<uint_fast64_t, 64, 312, 156, 31,
0xb502'6f5a'a966'19e9, 29, 0x5555'5555'5555'5555, 17,
0x71d6'7fff'eda6'0000, 37, 0xff7'eee0'0000'0000, 43, 6'364'136'223'846'793'005>;

4 Required behavior: The $10000^{th}$ consecutive invocation of a default-constructed object of type `mt19937_64` produces the value 9981545732273789042.
using ranlux24_base =
    subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>;
5  
Required behavior: The 10000th consecutive invocation of a default-constructed object of type ran-
lux24_base produces the value 7937952.

using ranlux48_base =
    subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>;
6  
Required behavior: The 10000th consecutive invocation of a default-constructed object of type ran-
lux48_base produces the value 61839128582725.

using ranlux24 = discard_block_engine<ranlux24_base, 223, 23>;
7  
Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux24
produces the value 9901578.

using ranlux48 = discard_block_engine<ranlux48_base, 389, 11>;
8  
Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux48
produces the value 249142670248501.

using knuth_b = shuffle_order_engine<minstd_rand0, 256>;
9  
Required behavior: The 10000th consecutive invocation of a default-constructed object of type knuth_b
produces the value 1112339016.

using default_random_engine = implementation-defined;
10  Remarks: The choice of engine type named by this typedef is implementation-defined.

[Note 1: The implementation can select this type on the basis of performance, size, quality, or any combina-
tion of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or
lightweight use. Because different implementations can select different underlying engine types, code that uses
this typedef need not generate identical sequences across implementations. — end note]

26.6.7 Class random_device

1  A random_device uniform random bit generator produces nondeterministic random numbers.

2  If implementation limitations prevent generating nondeterministic random numbers, the implementa-
tion may employ a random number engine.

class random_device {
    public:
        // types
        using result_type = unsigned int;

        // generator characteristics
        static constexpr result_type min() { return numeric_limits<result_type>::min(); }
        static constexpr result_type max() { return numeric_limits<result_type>::max(); }

        // constructors
        random_device() : random_device(implementation-defined) {}
        explicit random_device(const string& token);

        // generating functions
        result_type operator()();

        // property functions
        double entropy() const noexcept;

        // no copy functions
        random_device(const random_device&) = delete;
        void operator=(const random_device&) = delete;
    };

§ 26.6.7
explicit random_device(const string& token);

3   Throws: A value of an implementation-defined type derived from exception if the random_device could not be initialized.

4   Remarks: The semantics of the token parameter and the token value used by the default constructor are implementation-defined.250

double entropy() const noexcept;

5   Returns: If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate251 for the random numbers returned by operator(), in the range \( \min() \) to \( \log(\max() + 1) \).

result_type operator()();

6   Returns: A nondeterministic random value, uniformly distributed between \( \min() \) and \( \max() \) (inclusive). It is implementation-defined how these values are generated.

7   Throws: A value of an implementation-defined type derived from exception if a random number could not be obtained.

26.6.8 Utilities [rand.util]

26.6.8.1 Class seed_seq [rand.util.seedseq]

class seed_seq {
public:
   // types
   using result_type = uint_least32_t;

   // constructors
   seed_seq();
   template<class T>
   seed_seq(initializer_list<T> il);
   template<class InputIterator>
   seed_seq(InputIterator begin, InputIterator end);

   // generating functions
   template<class RandomAccessIterator>
   void generate(RandomAccessIterator begin, RandomAccessIterator end);

   // property functions
   size_t size() const noexcept;
   template<class OutputIterator>
   void param(OutputIterator dest) const;

   // no copy functions
   seed_seq(const seed_seq&) = delete;
   void operator=(const seed_seq&) = delete;

private:
   vector<result_type> v; // exposition only
};

seed_seq();

1   Postconditions: v.empty() is true.

2   Throws: Nothing.

3 template<class T>
   seed_seq(initializer_list<T> il);

3 Mandates: T is an integer type.

4 Effects: Same as seed_seq(il.begin(), il.end()).

250) The parameter is intended to allow an implementation to differentiate between different sources of randomness.
251) If a device has \( n \) states whose respective probabilities are \( P_0, \ldots, P_{n-1} \), the device entropy \( S \) is defined as

\[
S = - \sum_{i=0}^{n-1} P_i \cdot \log P_i.
\]
template<class InputIterator>
seed_seq(InputIterator begin, InputIterator end);

Mandates: iterator_traits<InputIterator>::value_type is an integer type.

Preconditions: InputIterator meets the Cpp17InputIterator requirements (23.3.5.3).

Effects: Initializes v by the following algorithm:
for (InputIterator s = begin; s != end; ++s)
v.push_back((s)mod232);

template<class RandomAccessIterator>
void generate(RandomAccessIterator begin, RandomAccessIterator end);

Mandates: iterator_traits<RandomAccessIterator>::value_type is an unsigned integer type capable of accommodating 32-bit quantities.

Preconditions: RandomAccessIterator meets the Cpp17RandomAccessIterator requirements (23.3.5.7) and the requirements of a mutable iterator.

Effects: Does nothing if begin == end. Otherwise, with s = v.size() and n = end - begin, fills the supplied range [begin,end) according to the following algorithm in which each operation is to be carried out modulo \(2^{32}\), each indexing operator applied to begin is to be taken modulo n, and \(T(x)\) is defined as \(x\ xor\ (x\ shift\ 27)\):

(10.1) — By way of initialization, set each element of the range to the value 0x8b8b8b8b. Additionally, for use in subsequent steps, let \(p = (n - t)/2\) and let \(q = p + t\), where
\[
t = (n \geq 623) \ ? \ (n \geq 68) \ ? \ 7 : (n \geq 39) \ ? \ 5 : (n \geq 7) \ ? \ 3 : (n - 1)/2;
\]

(10.2) — With \(m\) as the larger of \(s + 1\) and \(n\), transform the elements of the range: iteratively for \(k = 0,\ldots,m - 1\), calculate values
\[
\begin{align*}
r_1 &= 1664525 \cdot T(begin[k] \ xor \ begin[k + p] \ xor \ begin[k - 1]) \\
r_2 &= r_1 + \begin{cases} 
   s, & k = 0 \\
   k \mod n + v[k - 1], & 0 < k \leq s \\
   k \mod n, & s < k
\end{cases}
\end{align*}
\]
and, in order, increment begin[k+p] by \(r_1\), increment begin[k+q] by \(r_2\), and set begin[k] to \(r_2\).

(10.3) — Transform the elements of the range again, beginning where the previous step ended: iteratively for \(k = m,\ldots,m + n - 1\), calculate values
\[
\begin{align*}
r_3 &= 1566083941 \cdot T(begin[k] + begin[k + p] + begin[k - 1]) \\
r_4 &= r_3 - (k \mod n)
\end{align*}
\]
and, in order, update begin[k+p] by xoring it with \(r_3\), update begin[k+q] by xoring it with \(r_4\), and set begin[k] to \(r_4\).

Throws: What and when RandomAccessIterator operations of begin and end throw.

size_t size() const noexcept;

Returns: The number of 32-bit units that would be returned by a call to param().

Complexity: Constant time.

template<class OutputIterator>
void param(OutputIterator dest) const;

Mandates: Values of type result_type are writable (23.3.1) to dest.

Preconditions: OutputIterator meets the Cpp17OutputIterator requirements (23.3.5.4).

Effects: Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:
copy(v.begin(), v.end(), dest);

Throws: What and when OutputIterator operations of dest throw.
26.6.8.2 Function template `generate_canonical`  

```cpp
template<class RealType, size_t bits, class URBG>
RealType generate_canonical(URBG& g);
```

1. **Complexity:** Exactly
   \[ k = \max(1, \lceil b/\log_2 R \rceil) \]
   invocations of `g`, where \( b^{252} \) is the lesser of `numeric_limits<RealType>::digits` and `bits`, and \( R \) is the value of `g.max() - g.min() + 1`.

2. **Effects:** Invokes `g()` \( k \) times to obtain values \( g_0, \ldots, g_{k-1} \), respectively. Calculates a quantity
   \[
   S = \sum_{i=0}^{k-1} (g_i - g.min()) \cdot R^i
   \]
   using arithmetic of type `RealType`.

3. **Returns:** \( S/R^k \).

   [Note 1: \( 0 \leq S/R^k < 1 \). — end note]

4. **Throws:** What and when `g` throws.

5. [Note 2: If the values \( g_i \) produced by `g` are uniformly distributed, the instantiation’s results are distributed as uniformly as possible. Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random bit generator into a value that can be delivered by a random number distribution. — end note]

26.6.9 Random number distribution class templates

26.6.9.1 In general

1. Each type instantiated from a class template specified in this subclause 26.6.9 meets the requirements of a random number distribution (26.6.3.6) type.

2. Descriptions are provided in this subclause 26.6.9 only for distribution operations that are not described in 26.6.3.6 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

3. The algorithms for producing each of the specified distributions are implementation-defined.

4. The value of each probability density function \( p(z) \) and of each discrete probability function \( P(z_i) \) specified in this subclause is 0 everywhere outside its stated domain.

26.6.9.2 Uniform distributions

26.6.9.2.1 Class template `uniform_int_distribution`

1. A `uniform_int_distribution` random number distribution produces random integers \( i, a \leq i \leq b \), distributed according to the constant discrete probability function
   \[
   P(i | a, b) = 1/(b - a + 1).
   \]

```cpp
template<class IntType = int>
class uniform_int_distribution {
public:
   // types
   using result_type = IntType;
   using param_type = unspecified;

   // constructors and reset functions
   uniform_int_distribution() : uniform_int_distribution(0) {};
   explicit uniform_int_distribution(IntType a, IntType b = numeric_limits<IntType>::max());
   explicit uniform_int_distribution(const param_type& parm);
   void reset();

   // generating functions
   template<URBG>
   result_type operator()(URBG& g);
   template<URBG>
   result_type operator()(URBG& g, const param_type& parm);
}
```

252) \( b \) is introduced to avoid any attempt to produce more bits of randomness than can be held in `RealType`.  

§ 26.6.9.2.1
// property functions
result_type a() const;
result_type b() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit uniform_int_distribution(IntType a, IntType b = numeric_limits<IntType>::max());

2
Preconditions: a ≤ b.

3
Remarks: a and b correspond to the respective parameters of the distribution.

result_type a() const;
4
Returns: The value of the a parameter with which the object was constructed.

result_type b() const;
5
Returns: The value of the b parameter with which the object was constructed.

26.6.9.2.2 Class template uniform_real_distribution [rand.dist.uni.real]
A uniform_real_distribution random number distribution produces random numbers \( x, a \leq x < b \), distributed according to the constant probability density function

\[
p(x \mid a, b) = \frac{1}{b - a}.
\]

[Note 1: This implies that \( p(x \mid a, b) \) is undefined when \( a = b \). — end note]

template<class RealType = double>
class uniform_real_distribution {
public:
// types
using result_type = RealType;
using param_type = unspecified;

// constructors and reset functions
uniform_real_distribution() : uniform_real_distribution(0.0) {};
explicit uniform_real_distribution(RealType a, RealType b = 1.0);
explicit uniform_real_distribution(const param_type& parm);
void reset();

// generating functions
template<class URBG>
result_type operator()(URBG& g);
template<class URBG>
result_type operator()(URBG& g, const param_type& parm);

// property functions
result_type a() const;
result_type b() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit uniform_real_distribution(RealType a, RealType b = 1.0);
2
Preconditions: a ≤ b and \( b - a \leq \) numeric_limits<RealType>::max().
3
Remarks: a and b correspond to the respective parameters of the distribution.

result_type a() const;
4
Returns: The value of the a parameter with which the object was constructed.
Returns: The value of the b parameter with which the object was constructed.

26.6.9.3 Bernoulli distributions

26.6.9.3.1 Class bernoulli_distribution

A bernoulli_distribution random number distribution produces bool values b distributed according to the discrete probability function

\[ P(b|p) = \begin{cases} p & \text{if } b = \text{true}, \\ 1 - p & \text{if } b = \text{false} \end{cases} \]

```cpp
class bernoulli_distribution {
public:
    // types
    using result_type = bool;
    using param_type = unspecified;

    // constructors and reset functions
    bernoulli_distribution() : bernoulli_distribution(0.5) {};
    explicit bernoulli_distribution(double p);
    explicit bernoulli_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

    // property functions
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

2 Preconditions: \( 0 \leq p \leq 1 \).
3 Remarks: p corresponds to the parameter of the distribution.

double p() const;

Returns: The value of the p parameter with which the object was constructed.

26.6.9.3.2 Class template binomial_distribution

A binomial_distribution random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function

\[ P(i|t,p) = \binom{t}{i} \cdot p^i \cdot (1 - p)^{t-i} . \]

```cpp
template<class IntType = int>
class binomial_distribution {
public:
    // types
    using result_type = IntType;
    using param_type = unspecified;
};
```
// constructors and reset functions
binomial_distribution() : binomial_distribution(1) {}
explicit binomial_distribution(IntType t, double p = 0.5);
explicit binomial_distribution(const param_type& parm);
void reset();

// generating functions
template<class URBG>
result_type operator()(URBG& g);
template<class URBG>
result_type operator()(URBG& g, const param_type& parm);

// property functions
IntType t() const;
double p() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

explicit binomial_distribution(IntType t, double p = 0.5);

2
Preconditions: 0 ≤ p ≤ 1 and 0 ≤ t.

3
Remarks: t and p correspond to the respective parameters of the distribution.

IntType t() const;
Returns: The value of the t parameter with which the object was constructed.

double p() const;
Returns: The value of the p parameter with which the object was constructed.

26.6.9.3.3 Class template geometric_distribution
A geometric_distribution random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function
\[
P(i \mid p) = p \cdot (1 - p)^i.
\]

template<class IntType = int>
class geometric_distribution {
public:
// types
using result_type = IntType;
using param_type = unspecified;

// constructors and reset functions
geometric_distribution() : geometric_distribution(0.5) {}
explicit geometric_distribution(double p);
explicit geometric_distribution(const param_type& parm);
void reset();

// generating functions
template<class URBG>
result_type operator()(URBG& g);
template<class URBG>
result_type operator()(URBG& g, const param_type& parm);

// property functions
double p() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

};
explicit geometric_distribution(double p);

Preconditions: $0 < p < 1$.
 Remarks: $p$ corresponds to the parameter of the distribution.

double p() const;
 Returns: The value of the $p$ parameter with which the object was constructed.

26.6.9.3.4 Class template negative_binomial_distribution

A negative_binomial_distribution random number distribution produces random integers $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid k, p) = \binom{k + i - 1}{i} \cdot p^k \cdot (1 - p)^i.$$ 

[Note 1: This implies that $P(i \mid k, p)$ is undefined when $p == 1$. — end note]

template<class IntType = int>
class negative_binomial_distribution {
   public:
      // types
      using result_type = IntType;
      using param_type = unspecified;

      // constructor and reset functions
      negative_binomial_distribution() : negative_binomial_distribution(1) {}
      explicit negative_binomial_distribution(IntType k, double p = 0.5);
      explicit negative_binomial_distribution(const param_type& parm);
      void reset();

      // generating functions
      template<class URBG>
      result_type operator()(URBG& g);
      template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

      // property functions
      IntType k() const;
      double p() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
      result_type max() const;

   };

explicit negative_binomial_distribution(IntType k, double p = 0.5);

Preconditions: $0 < p \leq 1$ and $0 < k$.
 Remarks: $k$ and $p$ correspond to the respective parameters of the distribution.

IntType k() const;
 Returns: The value of the $k$ parameter with which the object was constructed.

double p() const;
 Returns: The value of the $p$ parameter with which the object was constructed.

26.6.9.4 Poisson distributions

26.6.9.4.1 Class template poisson_distribution

A poisson_distribution random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid \mu) = \frac{e^{-\mu} \mu^i}{i!}.$$ 

§ 26.6.9.4.1
The distribution parameter $\mu$ is also known as this distribution’s mean.

```cpp
template<class IntType = int>
class poisson_distribution
{
public:
    // types
    using result_type = IntType;
    using param_type = unspecified;

    // constructors and reset functions
    poisson_distribution() : poisson_distribution(1.0) {} explicit poisson_distribution(double mean);
    explicit poisson_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

    // property functions
    double mean() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit poisson_distribution(double mean);
```

2 Preconditions: $0 < \text{mean}$.  

3 Remarks: mean corresponds to the parameter of the distribution.  

4 double mean() const;

4 Returns: The value of the mean parameter with which the object was constructed.  

26.6.9.4.2 Class template exponential_distribution [rand.dist.pois.exp]  

1 An exponential_distribution random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid \lambda) = \lambda e^{-\lambda x}.$$  

```cpp
template<class RealType = double>
class exponential_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructors and reset functions
    exponential_distribution() : exponential_distribution(1.0) {} explicit exponential_distribution(RealType lambda);
    explicit exponential_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);
};
```
explicit exponential_distribution(RealType lambda);

Preconditions: $0 < \lambda$.

Remarks: $\lambda$ corresponds to the parameter of the distribution.

RealType lambda() const;

Returns: The value of the $\lambda$ parameter with which the object was constructed.

26.6.9.4.3 Class template gamma_distribution

A gamma_distribution random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x | \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^\alpha \cdot \Gamma(\alpha)} \cdot x^{\alpha-1}.$$
26.6.9.4.4 Class template weibull_distribution

A `weibull_distribution` random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function

\[
p(x | a, b) = \frac{a}{b} \cdot \left( \frac{x}{b} \right)^{a-1} \cdot \exp \left( - \left( \frac{x}{b} \right)^a \right).
\]

```
template<class RealType = double>
class weibull_distribution {
    // types
    using result_type = RealType;
    using param_type = unspecified;
    // constructor and reset functions
    weibull_distribution() : weibull_distribution(1.0) {}
    explicit weibull_distribution(RealType a, RealType b = 1.0);
    explicit weibull_distribution(const param_type& parm);
    void reset();
    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);
    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

2

Preconditions: \( 0 < a \) and \( 0 < b \).

Remarks: \( a \) and \( b \) correspond to the respective parameters of the distribution.

```
extreme_value_distribution
```

26.6.9.4.5 Class template extreme_value_distribution

An `extreme_value_distribution` random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x | a, b) = \frac{1}{b} \cdot \exp \left( \frac{a - x}{b} - \exp \left( \frac{a - x}{b} \right) \right).
\]

```
template<class RealType = double>
class extreme_value_distribution {
    // types
    using result_type = RealType;
    using param_type = unspecified;

253) The distribution corresponding to this probability density function is also known (with a possible change of variable) as the Gumbel Type I, the log-Weibull, or the Fisher-Tippett Type I distribution.

§ 26.6.9.4.5
## 26.6.9.5 Normal distributions

### 26.6.9.5.1 Class template normal_distribution

A `normal_distribution` random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x | \mu, \sigma) = \frac{1}{\sigma \sqrt{2 \pi}} \exp \left( -\frac{(x - \mu)^2}{2\sigma^2} \right)
\]

The distribution parameters \( \mu \) and \( \sigma \) are also known as this distribution’s mean and standard deviation.

```cpp
template<class RealType = double>
class normal_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructors and reset functions
    normal_distribution() : normal_distribution(0.0) {}  
    explicit normal_distribution(RealType mean, RealType stddev = 1.0);  
    explicit normal_distribution(const param_type& parm);  
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType mean() const;
    RealType stddev() const;
};
```
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

explicit normal_distribution(RealType mean, RealType stddev = 1.0);

Preconditions: \(0 < \text{stddev} \).
Remarks: \text{mean} and \text{stddev} correspond to the respective parameters of the distribution.

RealType mean() const;
Returns: The value of the mean parameter with which the object was constructed.

RealType stddev() const;
Returns: The value of the stddev parameter with which the object was constructed.

26.6.9.5.2 Class template lognormal_distribution

A lognormal_distribution random number distribution produces random numbers \(x > 0\) distributed according to the probability density function

\[
p(x | m, s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp\left(-\frac{(\ln x - m)^2}{2s^2}\right).
\]

template<class RealType = double>
class lognormal_distribution {
public:
  // types
  using result_type = RealType;
  using param_type = unspecified;

  // constructor and reset functions
  lognormal_distribution() : lognormal_distribution(0.0) {};
  explicit lognormal_distribution(RealType m, RealType s = 1.0);
  explicit lognormal_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URBG>
  result_type operator()(URBG& g);
  template<class URBG>
  result_type operator()(URBG& g, const param_type& parm);

  // property functions
  RealType m() const;
  RealType s() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};

explicit lognormal_distribution(RealType m, RealType s = 1.0);

Preconditions: \(0 < s\).
Remarks: \text{m} and \text{s} correspond to the respective parameters of the distribution.

RealType m() const;
Returns: The value of the \text{m} parameter with which the object was constructed.

RealType s() const;
Returns: The value of the \text{s} parameter with which the object was constructed.
26.6.9.5.3 Class template chi_squared_distribution

A chi_squared_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x \mid n) = \frac{x^{(n/2)-1} e^{-x/2}}{\Gamma(n/2) 2^{n/2}}.
\]

```cpp
template<class RealType = double>
class chi_squared_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructor and reset functions
    chi_squared_distribution() : chi_squared_distribution(1.0) {} // default constructor
    explicit chi_squared_distribution(RealType n);
    explicit chi_squared_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Preconditions: \( 0 < n \).

Remarks: \( n \) corresponds to the parameter of the distribution.

Returns: The value of the \( n \) parameter with which the object was constructed.

26.6.9.5.4 Class template cauchy_distribution

A cauchy_distribution random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x \mid a, b) = \left( \pi b \left( 1 + \left( \frac{x - a}{b} \right)^2 \right) \right)^{-1}.
\]

```cpp
template<class RealType = double>
class cauchy_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructor and reset functions
    cauchy_distribution() : cauchy_distribution(0.0) {} // default constructor
    explicit cauchy_distribution(RealType a, RealType b = 1.0);
    explicit cauchy_distribution(const param_type& parm);
    void reset();
};
```
// generating functions
template<class URBG>
    result_type operator()(URBG& g);
template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

// property functions
RealType a() const;
RealType b() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit cauchy_distribution(RealType a, RealType b = 1.0);

// Preconditions
0 < b.
Remarks: a and b correspond to the respective parameters of the distribution.

RealType a() const;
Returns: The value of the a parameter with which the object was constructed.

RealType b() const;
Returns: The value of the b parameter with which the object was constructed.

26.6.9.5.5 Class template fisher_f_distribution
A fisher_f_distribution random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$
p(x | m, n) = \frac{\Gamma((m + n)/2)}{\Gamma(m/2) \Gamma(n/2)} \cdot \left(\frac{m}{n}\right)^{m/2} \cdot x^{(m/2)-1} \cdot \left(1 + \frac{m x}{n}\right)^{-(m+n)/2}.
$$

template<class RealType = double>
    class fisher_f_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructor and reset functions
    fisher_f_distribution() : fisher_f_distribution(1.0) {} // Preconditions: 0 < m and 0 < n.
    explicit fisher_f_distribution(RealType m, RealType n = 1.0);
    explicit fisher_f_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
        result_type operator()(URBG& g);
    template<class URBG>
        result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType m() const;
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit fisher_f_distribution(RealType m, RealType n = 1);

Remarks: \( m \) and \( n \) correspond to the respective parameters of the distribution.

RealType \( m() \) const;

Returns: The value of the \( m \) parameter with which the object was constructed.

RealType \( n() \) const;

Returns: The value of the \( n \) parameter with which the object was constructed.

### 26.6.9.5.6 Class template student_t_distribution

[\textit{rand.dist.norm.t}]

A \textit{student_t_distribution} random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x \mid n) = \frac{1}{\sqrt{n\pi}} \frac{\Gamma((n+1)/2)}{\Gamma(n/2)} \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2}.
\]

template<class RealType = double>
class student_t_distribution {
public:
  // types
  using result_type = RealType;
  using param_type = unspecified;

  // constructor and reset functions
  student_t_distribution() : student_t_distribution(1.0) {}
  explicit student_t_distribution(RealType n);
  explicit student_t_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URBG>
  result_type operator()(URBG& g);
  template<class URBG>
  result_type operator()(URBG& g, const param_type& parm);

  // property functions
  RealType n() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};

explicit student_t_distribution(RealType n);

Preconditions: \( 0 < n \).

Remarks: \( n \) corresponds to the parameter of the distribution.

### 26.6.9.6 Sampling distributions

[\textit{rand.dist.samp}]

### 26.6.9.6.1 Class template discrete_distribution

[\textit{rand.dist.samp.discrete}]

A \textit{discrete_distribution} random number distribution produces random integers \( i \), \( 0 \leq i < n \), distributed according to the discrete probability function

\[
P(i \mid p_0, \ldots, p_{n-1}) = p_i.
\]

Unless specified otherwise, the distribution parameters are calculated as: \( p_k = w_k / S \) for \( k = 0, \ldots, n - 1 \), in which the values \( w_k \), commonly known as the \textit{weights}, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: \( 0 < S = w_0 + \cdots + w_{n-1} \).

\§ 26.6.9.6.1
template<class IntType = int>
class discrete_distribution {
public:

    // types
    using result_type = IntType;
    using param_type = unspecified;

    // constructor and reset functions
    discrete_distribution();
    template<class InputIterator>
    discrete_distribution(InputIterator firstW, InputIterator lastW);
    discrete_distribution(initializer_list<double> wl);
    template<class UnaryOperation>
    discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);
    explicit discrete_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

    // property functions
    vector<double> probabilities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

discrete_distribution();
3
Effects: Constructs a discrete_distribution object with \( n = 1 \) and \( p_0 = 1 \).

[Note 1: Such an object will always deliver the value 0. — end note]

template<class InputIterator>
    discrete_distribution(InputIterator firstW, InputIterator lastW);
4
Mandates: is_convertible_v<iterator_traits<InputIterator>::value_type, double> is true.
5
Preconditions: InputIterator meets the Cpp17InputIterator requirements (23.3.5.3). If firstW ==
lastW, let \( n = 1 \) and \( w_0 = 1 \). Otherwise, [firstW, lastW) forms a sequence \( w \) of length \( n > 0 \).
6
Effects: Constructs a discrete_distribution object with probabilities given by the formula above.

discrete_distribution(initializer_list<double> wl);
7
Effects: Same as discrete_distribution(wl.begin(), wl.end()).

template<class UnaryOperation>
    discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);
8
Mandates: is_invocable_r_v<double, UnaryOperation&, double> is true.
9
Preconditions: If \( nw = 0 \), let \( n = 1 \), otherwise let \( n = nw \). The relation \( 0 < \delta = (xmax - xmin)/n \) holds.
10
Effects: Constructs a discrete_distribution object with probabilities given by the formula above,
using the following values: If \( nw = 0 \), let \( w_0 = 1 \). Otherwise, let \( w_k = fw(xmin + k \cdot \delta + \delta/2) \) for
\( k = 0, \ldots, n - 1 \).

Complexity: The number of invocations of \( fw \) does not exceed \( n \).

vector<double> probabilities() const;
12
Returns: A vector<double> whose size member returns \( n \) and whose operator[] member returns \( p_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n - 1 \).
26.6.9.6.2 Class template piecewise_constant_distribution

A piecewise_constant_distribution random number distribution produces random numbers \( x, b_0 \leq x < b_n \), uniformly distributed over each subinterval \([b_i, b_{i+1})\) according to the probability density function

\[
p(x | b_0, \ldots, b_n, \rho_0, \ldots, \rho_{n-1}) = \rho_i, \quad \text{for} \quad b_i \leq x < b_{i+1}.
\]

The \( n + 1 \) distribution parameters \( b_i \), also known as this distribution's interval boundaries, shall satisfy the relation \( b_i < b_{i+1} \) for \( i = 0, \ldots, n - 1 \). Unless specified otherwise, the remaining \( n \) distribution parameters are calculated as:

\[
p_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \quad \text{for} \quad k = 0, \ldots, n - 1,
\]

in which the values \( w_k \), commonly known as the weights, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: \( 0 < S = w_0 + \cdots + w_{n-1} \).

template<class RealType = double>
class piecewise_constant_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructor and reset functions
    piecewise_constant_distribution();
    template<class InputIteratorB, class InputIteratorW>
        piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
            InputIteratorW firstW);
    template<class UnaryOperation>
        piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
    template<class UnaryOperation>
        piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax,
            UnaryOperation fw);
    explicit piecewise_constant_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
        result_type operator()(URBG& g);
    template<class URBG>
        result_type operator()(URBG& g, const param_type& parm);

    // property functions
    vector<result_type> intervals() const;
    vector<result_type> densities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

Effects: Constructs a piecewise_constant_distribution object with \( n = 1, \rho_0 = 1, b_0 = 0, \) and \( b_1 = 1 \).

Mandates: Both of

(4.1) \( \text{is_convertible_v<iterator_traits<InputIteratorB>::value_type, double> \ } \)
(4.2) \( \text{is_convertible_v<iterator_traits<InputIteratorW>::value_type, double> \ } \)
Preconditions: InputIteratorB and InputIteratorW each meet the C++17 InputIterator requirements (23.3.5.3). If firstB == lastB or ++firstB == lastB, let \( n = 1, w_0 = 1, b_0 = 0, \) and \( b_1 = 1 \). Otherwise, \([\text{firstB, lastB}]\) forms a sequence \( b \) of length \( n + 1 \), the length of the sequence \( w \) starting from firstW is at least \( n \), and any \( w_k \) for \( k \geq n \) are ignored by the distribution.

Effects: Constructs a piecewise_constant_distribution object with parameters as specified above.

```
template<class UnaryOperation>
piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
```

Mandates: is_invocable_r_v<double, UnaryOperation&, double> is true.

Effects: Constructs a piecewise_constant_distribution object with parameters taken or calculated from the following values: If \( \text{bl.size()} < 2 \), let \( n = 1, w_0 = 1, b_0 = 0, \) and \( b_1 = 1 \). Otherwise, let \([\text{bl.begin()}, \text{bl.end()}]\) form a sequence \( b_0, ..., b_n \), and let \( w_k = \text{fw}((b_{k+1}+b_k)/2) \) for \( k = 0, ..., n-1 \).

Complexity: The number of invocations of \( \text{fw} \) does not exceed \( n \).

```
template<class UnaryOperation>
piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
```

Mandates: is_invocable_r_v<double, UnaryOperation&, double> is true.

Preconditions: If \( \text{nw} = 0 \), let \( n = 1 \), otherwise let \( n = \text{nw} \). The relation \( 0 < \delta = (\text{xmax} - \text{xmin})/n \) holds.

Effects: Constructs a piecewise_constant_distribution object with parameters taken or calculated from the following values: Let \( b_k = \text{xmin} + k \cdot \delta \) for \( k = 0, ..., n \), and \( w_k = \text{fw}(b_k + \delta/2) \) for \( k = 0, ..., n-1 \).

Complexity: The number of invocations of \( \text{fw} \) does not exceed \( n \).

```
vector<result_type> intervals() const;
```

Returns: A vector<result_type> whose size member returns \( n + 1 \) and whose operator[] member returns \( b_k \) when invoked with argument \( k \) for \( k = 0, ..., n \).

```
vector<result_type> densities() const;
```

Returns: A vector<result_type> whose size member returns \( n \) and whose operator[] member returns \( \rho_k \) when invoked with argument \( k \) for \( k = 0, ..., n - 1 \).

### 26.6.9.6.3 Class template piecewise_linear_distribution [rand.dist.samp.plinear]

A piecewise_linear_distribution random number distribution produces random numbers \( x, b_0 \leq x < b_n \), distributed over each subinterval \([b_i, b_{i+1})\) according to the probability density function

\[
p(x \mid b_0, \ldots, b_n; \rho_0, \ldots, \rho_n) = \rho_i \cdot \frac{b_{i+1} - x}{b_{i+1} - b_i} + \frac{x - b_i}{b_{i+1} - b_i}, \quad \text{for } b_i \leq x < b_{i+1}.
\]

The \( n + 1 \) distribution parameters \( b_i \), also known as this distribution’s internal boundaries, shall satisfy the relation \( b_i < b_{i+1} \) for \( i = 0, \ldots, n - 1 \). Unless specified otherwise, the remaining \( n + 1 \) distribution parameters are calculated as \( \rho_k = w_k/S \) for \( k = 0, \ldots, n \), in which the values \( w_k \), commonly known as the weights at boundaries, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold:

\[
0 < S = \frac{1}{2} \sum_{k=0}^{n-1} (w_k + w_{k+1}) \cdot (b_{k+1} - b_k).
\]

```
template<class RealType = double>
class piecewise_linear_distribution {
    public:
        // types
        using result_type = RealType;
        using param_type = unspecified;

        // constructor and reset functions
        piecewise_linear_distribution();
        piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB, InputIteratorW firstW);

    § 26.6.9.6.3 1208
```
template<class UnaryOperation>
    piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);

template<class UnaryOperation>
    piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);

explicit piecewise_linear_distribution(const param_type& parm);
void reset();

// generating functions
template<class URBG>
    result_type operator()(URBG& g);

template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

// property functions
vector<result_type> intervals() const;
vector<result_type> densities() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

};

piecewise_linear_distribution();

Effects: Constructs a piecewise_linear_distribution object with \( n = 1, \rho_0 = \rho_1 = 1, b_0 = 0, \) and 
\( b_1 = 1. \)

template<class InputIteratorB, class InputIteratorW>
    piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                          InputIteratorW firstW);

Mandates: is_invocable_r_v<double, UnaryOperation&, double> is true.

Preconditions: If \( \text{firstB} == \text{lastB} \) or ++firstB == lastB, let \( n = 1, \rho_0 = \rho_1 = 1, b_0 = 0, \) and 
\( b_1 = 1. \)
Otherwise, \([\text{firstB}, \text{lastB}]\) forms a sequence \( b \) of length \( n + 1, \) the length of the sequence \( w \) starting 
from \( \text{firstW} \) is at least \( n + 1, \) and any \( w_k \) for \( k \geq n + 1 \) are ignored by the distribution.

Effects: Constructs a piecewise_linear_distribution object with parameters as specified above.

template<class UnaryOperation>
    piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);

Mandates: is_invocable_r_v<double, UnaryOperation&, double> is true.

Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated 
from the following values: If \( \text{bl.size()} < 2, \) let \( n = 1, \rho_0 = \rho_1 = 1, b_0 = 0, \) and 
\( b_1 = 1. \) Otherwise, let \([\text{bl.begin()}, \text{bl.end()}]\) form a sequence \( b_0, \ldots, b_n, \) and let \( w_k = \text{fw}(b_k) \) for \( k = 0, \ldots, n. \)

Complexity: The number of invocations of \( \text{fw} \) does not exceed \( n + 1. \)

template<class UnaryOperation>
    piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);

Mandates: is_invocable_r_v<double, UnaryOperation&, double> is true.

Preconditions: If \( \text{nw} = 0, \) let \( n = 1, \) otherwise let \( n = \text{nw}. \) The relation \( 0 < \delta = (\text{xmax} - \text{xmin})/n \) holds.

Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated 
from the following values: Let \( b_k = \text{xmin} + k \cdot \delta \) for \( k = 0, \ldots, n, \) and \( w_k = \text{fw}(b_k) \) for \( k = 0, \ldots, n. \)

Complexity: The number of invocations of \( \text{fw} \) does not exceed \( n + 1. \)

vector<result_type> intervals() const;

Returns: A vector<result_type> whose size member returns \( n + 1 \) and whose operator[] member 
returns \( b_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n. \)
vector<result_type> densities() const;

Returns: A vector<result_type> whose size member returns n and whose operator[] member returns \( \rho_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n \).

26.6.10 Low-quality random number generation

[Note 1: The header `<cstdlib>` (17.2.2) declares the functions described in this subclause. — end note]

int rand();
void srand(unsigned int seed);

Effects: The rand and srand functions have the semantics specified in the C standard library.

Remarks: The implementation may specify that particular library functions may call rand. It is implementation-defined whether the rand function may introduce data races (16.4.6.10).

[Note 2: The other random number generation facilities in this document (26.6) are often preferable to rand, because rand’s underlying algorithm is unspecified. Use of rand therefore continues to be non-portable, with unpredictable and oft-questionable quality and performance. — end note]

See also: ISO C 7.22.2

26.7 Numeric arrays

26.7.1 Header `<valarray>` synopsis

#include <initializer_list>

namespace std {

template<class T> class valarray; // An array of type T
class slice; // a BLAS-like slice out of an array
template<class T> class slice_array;
template<class T> class gslice; // a generalized slice out of an array
template<class T> class gslice_array;
template<class T> class mask_array; // a masked array
template<class T> class indirect_array; // an indirected array

template<class T> void swap(valarray<T>&, valarray<T>&) noexcept;

template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator* (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> operator* (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator/ (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> operator/ (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator% (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> operator% (const typename valarray<T>::value_type&, const valarray<T>&);

§ 26.7.1
template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator^ (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> operator^ (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator& (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> operator& (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator| (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> operator| (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> operator<< (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator<< (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> operator<< (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> operator>> (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator>> (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> operator>>(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator&&(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator&&(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator&&(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator||(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator||(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator||(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator==(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator==(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator==(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator!=(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator!=(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator< (const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator< (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator< (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator> (const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator> (const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator> (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator<=(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator<=(const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator<=(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator>=(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> abs (const valarray<T>&);

template<class T> valarray<T> acos (const valarray<T>&);

template<class T> valarray<T> asin (const valarray<T>&);

template<class T> valarray<T> atan (const valarray<T>&);

template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> atan2(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> atan2(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> cos (const valarray<T>&);

template<class T> valarray<T> cosh (const valarray<T>&);

template<class T> valarray<T> exp (const valarray<T>&);

template<class T> valarray<T> log (const valarray<T>&);

template<class T> valarray<T> log10(const valarray<T>&);

template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> pow(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> pow(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> sin (const valarray<T>&);

template<class T> valarray<T> sinh (const valarray<T>&);

template<class T> valarray<T> sqrt (const valarray<T>&);

template<class T> valarray<T> tan (const valarray<T>&);

template<class T> valarray<T> tanh (const valarray<T>&);

template<class T> unspecified1 begin(valarray<T>& v);

template<class T> unspecified2 begin(const valarray<T>& v);

template<class T> unspecified1 end(valarray<T>& v);

template<class T> unspecified2 end(const valarray<T>& v);

The header `<valarray>` defines five class templates (`valarray`, `slice_array`, `gslice_array`, `mask_array`, and `indirect_array`), two classes (`slice` and `gslice`), and a series of related function templates for representing and manipulating arrays of values.

The `valarray` array classes are defined to be free of certain forms of aliasing, thus allowing operations on these classes to be optimized.

Any function returning a `valarray<T>` is permitted to return an object of another type, provided all the const member functions of `valarray<T>` are also applicable to this type. This return type shall not add more than two levels of template nesting over the most deeply nested argument type.254

Implementations introducing such replacement types shall provide additional functions and operators as follows:

- for every function taking a `const valarray<T>&` other than `begin` and `end` (26.7.10), identical functions taking the replacement types shall be added;
- for every function taking two `const valarray<T>&` arguments, identical functions taking every combination of `const valarray<T>&` and replacement types shall be added.

254) Annex B recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.
In particular, an implementation shall allow a `valarray<T>` to be constructed from such replacement types and shall allow assignments and compound assignments of such types to `valarray<T>`, `slice_array<T>`, `gslice_array<T>`, `mask_array<T>` and `indirect_array<T>` objects.

These library functions are permitted to throw a `bad_alloc` (17.6.4.1) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

### 26.7.2 Class template `valarray`

#### 26.7.2.1 Overview

```cpp
namespace std {
    template<class T> class valarray {
        using value_type = T;

        // 26.7.2.2, construct/destroy
        valarray();
        explicit valarray(size_t);
        valarray(const T&, size_t);
        valarray(const T*, size_t);
        valarray(const valarray&);
        valarray(valarray&&) noexcept;
        valarray(const slice_array<T>&);
        valarray(const gslice_array<T>&);
        valarray(const mask_array<T>&);
        valarray(const indirect_array<T>&);
        valarray(initializer_list<T>);
        ~valarray();

        // 26.7.2.3, assignment
        valarray& operator=(const valarray&);
        valarray& operator=(valarray&&) noexcept;
        valarray& operator=(initializer_list<T>);
        valarray& operator=(const T&);
        valarray& operator=(const slice_array<T>&);
        valarray& operator=(const gslice_array<T>&);
        valarray& operator=(const mask_array<T>&);
        valarray& operator=(const indirect_array<T>&);

        // 26.7.2.4, element access
        const T& operator[](size_t) const;
        T& operator[](size_t);

        // 26.7.2.5, subset operations
        valarray operator[](slice) const;
        slice_array<T> operator[](slice);
        valarray operator[](const gslice&);
        gslice_array<T> operator[](const gslice&);
        valarray operator[](const valarray<bool>&) const;
        mask_array<T> operator[](const valarray<bool>&);
        valarray operator[](const valarray<size_t>&) const;
        indirect_array<T> operator[](const valarray<size_t>&);

        // 26.7.2.6, unary operators
        valarray operator+() const;
        valarray operator-() const;
        valarray operator~() const;
        valarray<bool> operator!() const;

        // 26.7.2.7, compound assignment
        valarray& operator*= (const T&);
        valarray& operator/= (const T&);
        valarray& operator%= (const T&);
        valarray& operator+= (const T&);
    }
}
```
The class template `valarray<T>` is a one-dimensional smart array, with elements numbered sequentially from zero. It is a representation of the mathematical concept of an ordered set of values. For convenience, an object of type `valarray<T>` is referred to as an “array” throughout the remainder of 26.7. The illusion of higher dimensionality may be produced by the familiar idiom of computed indices, together with the powerful subsetting capabilities provided by the generalized subscript operators.\footnote{The intent is to specify an array template that has the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporary objects. Thus, the `valarray` template is neither a matrix class nor a field class. However, it is a very useful building block for designing such classes.}

### 26.7.2.2 Constructors

#### `[valarray.cons]`

**valarray();**

*Effects*: Constructs a `valarray` that has zero length.\footnote{This default constructor is essential, since arrays of `valarray` can be useful. After initialization, the length of an empty array can be increased with the `resize` member function.}

**explicit valarray(size_t n);**

*Effects*: Constructs a `valarray` that has length \( n \). Each element of the array is value-initialized (9.4).

**valarray(const T& v, size_t n);**

*Effects*: Constructs a `valarray` that has length \( n \). Each element of the array is initialized with \( v \).

**valarray(const T* p, size_t n);**

*Preconditions*: \([p, p + n)\) is a valid range.
Effects: Constructs a `valarray` that has length n. The values of the elements of the array are initialized with the first n values pointed to by the first argument.

valarray(const valarray& v);

Effects: Constructs a `valarray` that has the same length as v. The elements are initialized with the values of the corresponding elements of v.

valarray(valarray&& v) noexcept;

Effects: Constructs a `valarray` that has the same length as v. The elements are initialized with the values of the corresponding elements of v.

Complexity: Constant.

valarray(initializer_list<T> il);

Effects: Equivalent to `valarray(il.begin(), il.size())`.

valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);

These conversion constructors convert one of the four reference templates to a `valarray`.

~valarray();

Effects: The destructor is applied to every element of *this; an implementation may return all allocated memory.

26.7.2.3 Assignment

valarray& operator=(const valarray& v);

Effects: Each element of the *this array is assigned the value of the corresponding element of v. If the length of v is not equal to the length of *this, resizes *this to make the two arrays the same length, as if by calling resize(v.size()), before performing the assignment.

Postconditions: size() == v.size().

Returns: *this.

valarray& operator=(valarray&& v) noexcept;

Effects: *this obtains the value of v. The value of v after the assignment is not specified.

Returns: *this.

Complexity: Linear.

valarray& operator=(initializer_list<T> il);

Effects: Equivalent to: return *this = valarray(il);

valarray& operator=(const T& v);

Effects: Assigns v to each element of *this.

Returns: *this.

valarray& operator=(const slice_array<T>&);
valarray& operator=(const gslice_array<T>&);
valarray& operator=(const mask_array<T>&);
valarray& operator=(const indirect_array<T>&);

Preconditions: The length of the array to which the argument refers equals size(). The value of an element in the left-hand side of a `valarray` assignment operator does not depend on the value of another element in that left-hand side.

257) This constructor is the preferred method for converting a C array to a `valarray` object.

258) This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they would need to implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.

§ 26.7.2.3 1215
These operators allow the results of a generalized subscripting operation to be assigned directly to a valarray.

26.7.2.4 Element access

const T& operator[](size_t n) const;
T& operator[](size_t n);

1 Preconditions: \( n < \text{size()} \) is true.
2 Returns: A reference to the corresponding element of the array.

\[\text{Note 1: The expression } (\text{a}[i] = q, \text{a}[i]) == q \text{ evaluates to true for any non-constant valarray<T> a, any } T q, \text{ and for any } size_t i \text{ such that the value of } i \text{ is less than the length of a.} \quad \text{— end note}\]
3 Remarks: The expression addressof(a[i+j]) == addressof(a[i]) + j evaluates to true for all size_t i and size_t j such that \( i+j < \text{a.size()} \).

The expression addressof(a[i]) != addressof(b[j]) evaluates to true for any two arrays a and b and for any size_t i and size_t j such that \( i < \text{a.size()} \) and \( j < \text{b.size()} \).

\[\text{Note 2: This property indicates an absence of aliasing and can be used to advantage by optimizing compilers. Compilers can take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from operator new, and other techniques to generate efficient valarrays.} \quad \text{— end note}\]

The reference returned by the subscript operator for an array shall be valid until the member function resize(size_t, T) (26.7.2.8) is called for that array or until the lifetime of that array ends, whichever happens first.

26.7.2.5 Subset operations

1 The member operator[] is overloaded to provide several ways to select sequences of elements from among those controlled by *this. Each of these operations returns a subset of the array. The const-qualified versions return this subset as a new valarray object. The non-const versions return a class template object which has reference semantics to the original array, working in conjunction with various overloads of operator= and other assigning operators to allow selective replacement (slicing) of the controlled sequence. In each case the selected element(s) shall exist.

valarray operator[](slice slicearr) const;

2 Returns: A valarray containing those elements of the controlled sequence designated by slicearr.

\[\text{Example 1: } \]
const valarray<char> v0("abcdefghiklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
— end example

slice_array<T> operator[](slice slicearr);

3 Returns: An object that holds references to elements of the controlled sequence selected by slicearr.

\[\text{Example 2: } \]
valarray<char> v0("abcdefghiklmnop", 16);
valarray<char> v1("ABCDE", 5);
v0[slice(2, 5, 3)] = v1;
// v0 == valarray<char>("abAdeBghCjkDmnEp", 16);
— end example

valarray operator[](const gslice& gslicearr) const;

4 Returns: A valarray containing those elements of the controlled sequence designated by gslicearr.

\[\text{Example 3: } \]
const valarray<char> v0("abcdefghiklmnop", 16);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };
const valarray<size_t> len(lv, 2), str(dv, 2);
// v0[gslice(3, len, str)] returns
// valarray<char>("dfhkmo", 6)
— end example

§ 26.7.2.5 1216
galice_array<T> operator[](const gslice& gslicearr);

Returns: An object that holds references to elements of the controlled sequence selected by gslicearr.

[Example 4:
  valarray<char> v0("abcdefghijklmnop", 16);
  valarray<char> v1("ABCDEF", 6);
  const size_t lv[] = { 2, 3 };
  const size_t dv[] = { 7, 2 };
  const valarray<size_t> len(lv, 2), str(dv, 2);
  v0[gslice(3, len, str)] = v1;
  // v0 == valarray<char>("abcAeBgCijDlEnFp", 16)
  — end example]

valarray operator[](const valarray<bool>& boolarr) const;

Returns: A valarray containing those elements of the controlled sequence designated by boolarr.

[Example 5:
  const valarray<char> v0("abcdefghijklmnop", 16);
  const bool vb[] = { false, false, true, true, false, true };
  // v0[valarray<bool>(vb, 6)] returns
  // valarray<char>("cdf", 3)
  — end example]

mask_array<T> operator[](const valarray<bool>& boolarr);

Returns: An object that holds references to elements of the controlled sequence selected by boolarr.

[Example 6:
  valarray<char> v0("abcdefghijklmnop", 16);
  valarray<char> v1("ABC", 3);
  const bool vb[] = { false, false, true, true, false, true };
  v0[valarray<bool>(vb, 6)] = v1;
  // v0 == valarray<char>("abABeCghijklmnop", 16)
  — end example]

valarray operator[](const valarray<size_t>& indarr) const;

Returns: A valarray containing those elements of the controlled sequence designated by indarr.

[Example 7:
  const valarray<char> v0("abcdefghijklmnop", 16);
  const size_t vi[] = { 7, 5, 2, 3, 8 };
  // v0[valarray<size_t>(vi, 5)] returns
  // valarray<char>("hfcdi", 5)
  — end example]

indirect_array<T> operator[](const valarray<size_t>& indarr);

Returns: An object that holds references to elements of the controlled sequence selected by indarr.

[Example 8:
  valarray<char> v0("abcdefghijklmnop", 16);
  valarray<char> v1("ABCDE", 5);
  const size_t vi[] = { 7, 5, 2, 3, 8 };
  v0[indirect_array<size_t>(vi, 5)] = v1;
  // v0 == valarray<char>("abCDeBGhijklmnop", 16)
  — end example]

26.7.2.6 Unary operators

valarray operator+() const;
valarray operator-() const;
valarray operator-() const;
valarray<bool> operator!() const;

Mandates: The indicated operator can be applied to operands of type T and returns a value of type T (bool for operator!) or which may be unambiguously implicitly converted to type T (bool for operator!).

Returns: A valarray whose length is size(). Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

26.7.2.7 Compound assignment

valarray& operator*= (const valarray& v);
valarray& operator/= (const valarray& v);
valarray& operator%= (const valarray& v);
valarray& operator+= (const valarray& v);
valarray& operator-= (const valarray& v);
valarray& operator&= (const valarray& v);
valarray& operator|= (const valarray& v);
valarray& operator<<=(const valarray& v);
valarray& operator>>=(const valarray& v);

Mandates: The indicated operator can be applied to two operands of type T.

Preconditions: size() == v.size() is true.

The value of an element in the left-hand side of a valarray compound assignment operator does not depend on the value of another element in that left hand side.

Effects: Each of these operators performs the indicated operation on each of the elements of *this and the corresponding element of v.

Returns: *this.

Remarks: The appearance of an array on the left-hand side of a compound assignment does not invalidate references or pointers.

valarray& operator*= (const T& v);
valarray& operator/= (const T& v);
valarray& operator%= (const T& v);
valarray& operator+= (const T& v);
valarray& operator-= (const T& v);
valarray& operator&= (const T& v);
valarray& operator|= (const T& v);
valarray& operator<<=(const T& v);
valarray& operator>>=(const T& v);

Mandates: The indicated operator can be applied to two operands of type T.

Effects: Each of these operators applies the indicated operation to each element of *this and v.

Returns: *this.

Remarks: The appearance of an array on the left-hand side of a compound assignment does not invalidate references or pointers to the elements of the array.

26.7.2.8 Member functions

void swap(valarray& v) noexcept;

Effects: *this obtains the value of v. v obtains the value of *this.

Complexity: Constant.

size_t size() const;

Returns: The number of elements in the array.

Complexity: Constant time.
T sum() const;

Mandates: operator+= can be applied to operands of type T.

Preconditions: size() > 0 is true.

Returns: The sum of all the elements of the array. If the array has length 1, returns the value of element 0. Otherwise, the returned value is calculated by applying operator+= to a copy of an element of the array and all other elements of the array in an unspecified order.

T min() const;

Preconditions: size() > 0 is true.

Returns: The minimum value contained in *this. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator<.

T max() const;

Preconditions: size() > 0 is true.

Returns: The maximum value contained in *this. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator<.

valarray shift(int n) const;

Returns: A valarray of length size(), each of whose elements I is (*this)[I + n] if I + n is non-negative and less than size(), otherwise T().

[Note 1: If element zero is taken as the leftmost element, a positive value of n shifts the elements left n places, with zero fill. — end note]

[Example 1: If the argument has the value -2, the first two elements of the result will be value-initialized (9.4); the third element of the result will be assigned the value of the first element of the argument; etc. — end example]

valarray cshift(int n) const;

Returns: A valarray of length size() that is a circular shift of *this. If element zero is taken as the leftmost element, a non-negative value of n shifts the elements circularly left n places and a negative value of n shifts the elements circularly right −n places.

valarray apply(T func(T)) const;
valarray apply(T func(const T&)) const;

Returns: A valarray whose length is size(). Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of *this.

void resize(size_t sz, T c = T());

Effects: Changes the length of the *this array to sz and then assigns to each element the value of the second argument. Resizing invalidates all pointers and references to elements in the array.

26.7.3 valarray non-member operations

26.7.3.1 Binary operators

template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>> (const valarray<T>&, const valarray<T>&);

Mandates: The indicated operator can be applied to operands of type T and returns a value of type T or which can be unambiguously implicitly converted to T.

Preconditions: The argument arrays have the same length.
Returns: A valarray whose length is equal to the length of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

```cpp
template<class T> valarray<T> operator* (const valarray<T>&, const typename valarray<T>::value_type&);
```

Mandates: The indicated operator can be applied to operands of type T and returns a value of type T or which can be unambiguously implicitly converted to T.

Returns: A valarray whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the non-array argument.

26.7.3.2 Logical operators

```cpp
template<class T> valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
```

Mandates: The indicated operator can be applied to operands of type T and returns a value of type bool or which can be unambiguously implicitly converted to bool.
Preconditions: The two array arguments have the same length.

Returns: A valarray<bool> whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

```
template<class T> valarray<bool> operator==(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator<(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator>(const valarray<T>&, const typename valarray<T>::value_type&);
```

Mandates: The indicated operator can be applied to operands of type T and returns a value of type bool or which can be unambiguously implicitly converted to bool.

Returns: A valarray<bool> whose length is equal to the length of the argument array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array and the non-array argument.

26.7.3.3 Transcendentals

```
template<class T> valarray<T> abs (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> atan2(const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<T> cos (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);
template<class T> valarray<T> exp (const valarray<T>&);
template<class T> valarray<T> log (const valarray<T>&);
template<class T> valarray<T> log10(const valarray<T>&);
template<class T> valarray<T> pow (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> pow (const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> pow (const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<T> sinh (const valarray<T>&);
template<class T> valarray<T> sqrt (const valarray<T>&);
```
```cpp
template<class T> valarray<T> tan (const valarray<T>&);
template<class T> valarray<T> tanh (const valarray<T>&);
```

**Mandates:** A unique function with the indicated name can be applied (unqualified) to an operand of type `T`. This function returns a value of type `T` or which can be unambiguously implicitly converted to type `T`.

### § 26.7.3.4 Specialized algorithms

```cpp
template<class T> void swap(valarray<T>& x, valarray<T>& y) noexcept;
```

**Effects:** Equivalent to `x.swap(y)`.

### § 26.7.4 Class slice

#### § 26.7.4.1 Overview

```cpp
namespace std {
  class slice {
    public:
      slice();
      slice(size_t, size_t, size_t);

      size_t start() const;
      size_t size() const;
      size_t stride() const;

      friend bool operator==(const slice& x, const slice& y);
  };
}
```

The `slice` class represents a BLAS-like slice from an array. Such a slice is specified by a starting index, a length, and a stride.\(^{259}\)

#### § 26.7.4.2 Constructors

```cpp
slice();
slice(size_t start, size_t length, size_t stride);
slice(const slice&);
```

The default constructor is equivalent to `slice(0, 0, 0)`. A default constructor is provided only to permit the declaration of arrays of slices. The constructor with arguments for a slice takes a start, length, and stride parameter.

**Example 1:** `slice(3, 8, 2)` constructs a slice which selects elements `3, 5, 7, ..., 17` from an array. — end example

#### § 26.7.4.3 Access functions

```cpp
size_t start() const;
size_t size() const;
size_t stride() const;
```

**Returns:** The start, length, or stride specified by a `slice` object.

**Complexity:** Constant time.

#### § 26.7.4.4 Operators

```cpp
friend bool operator==(const slice& x, const slice& y);
```

**Effects:** Equivalent to:

```cpp
return x.start() == y.start() && x.size() == y.size() && x.stride() == y.stride();
```

\(^{259}\) BLAS stands for *Basic Linear Algebra Subprograms*. C++ programs can instantiate this class. See, for example, Dongarra, Du Croz, Duff, and Hammarling: *A set of Level 3 Basic Linear Algebra Subprograms*; Technical Report MCS-P1-0888, Argonne National Laboratory (USA), Mathematics and Computer Science Division, August, 1988.
26.7.5 Class template slice_array

26.7.5.1 Overview

 namespace std {
 template<class T> class slice_array {
 public:
  using value_type = T;

  void operator=(const valarray<T>&) const;
  void operator**(const valarray<T>&) const;
  void operator/=(const valarray<T>&) const;
  void operator%=(const valarray<T>&) const;
  void operator+=(const valarray<T>&) const;
  void operator-=(const valarray<T>&) const;
  void operator^=(const valarray<T>&) const;
  void operator&=(const valarray<T>&) const;
  void operator|=(const valarray<T>&) const;
  void operator<<=(const valarray<T>&) const;
  void operator>>=(const valarray<T>&) const;

  slice_array(const slice_array&);
  ~slice_array();
  const slice_array& operator=(const slice_array&) const;
  void operator=(const T&) const;

  slice_array() = delete; // as implied by declaring copy constructor above
};
}

1 This template is a helper template used by the slice subscript operator

   slice_array<T> valarray<T>::operator[](slice);

2 It has reference semantics to a subset of an array specified by a slice object.

   [Example 1: The expression a[slice(1, 5, 3)] = b; has the effect of assigning the elements of b to a slice of the
   elements in a. For the slice shown, the elements selected from a are 1, 4, . . . , 13. — end example]

26.7.5.2 Assignment

 void operator=(const valarray<T>&) const;
 const slice_array& operator=(const slice_array&) const;

1 These assignment operators have reference semantics, assigning the values of the argument array
   elements to selected elements of the valarray<T> object to which the slice_array object refers.

26.7.5.3 Compound assignment

 void operator**(const valarray<T>&) const;
 void operator/=(const valarray<T>&) const;
 void operator%=(const valarray<T>&) const;
 void operator+=(const valarray<T>&) const;
 void operator-=(const valarray<T>&) const;
 void operator^=(const valarray<T>&) const;
 void operator&=(const valarray<T>&) const;
 void operator|=(const valarray<T>&) const;
 void operator<<=(const valarray<T>&) const;
 void operator>>=(const valarray<T>&) const;

1 These compound assignments have reference semantics, applying the indicated operation to the elements
   of the argument array and selected elements of the valarray<T> object to which the slice_array
   object refers.

26.7.5.4 Fill function

 void operator=(const T&) const;

1 This function has reference semantics, assigning the value of its argument to the elements of the
   valarray<T> object to which the slice_array object refers.
This class represents a generalized slice out of an array. A \texttt{gslice} is defined by a starting offset ($s$), a set of lengths ($l_j$), and a set of strides ($d_j$). The number of lengths shall equal the number of strides. A \texttt{gslice} represents a mapping from a set of indices ($i_j$), equal in number to the number of strides, to a single index $k$. It is useful for building multidimensional array classes using the \texttt{valarray} template, which is one-dimensional. The set of one-dimensional index values specified by a \texttt{gslice} are

$$ k = s + \sum_j i_j d_j $$

where the multidimensional indices $i_j$ range in value from 0 to $l_{ij} - 1$.

\textbf{Example 1:} The \texttt{gslice} specification

\begin{verbatim}
start = 3
length = (2, 4, 3)
stride = (19, 4, 1)
\end{verbatim}

yields the sequence of one-dimensional indices

$$ k = 3 + (0, 1) \times 19 + (0, 1, 2, 3) \times 4 + (0, 1, 2) \times 1 $$

which are ordered as shown in the following table:

\begin{verbatim}
(i0, i1, i2, k) =
(0, 0, 0, 3),
(0, 0, 1, 4),
(0, 0, 2, 5),
(0, 1, 0, 7),
(0, 1, 1, 8),
(0, 1, 2, 9),
(0, 2, 0, 11),
(0, 2, 1, 12),
(0, 2, 2, 13),
(0, 3, 0, 15),
(0, 3, 1, 16),
(0, 3, 2, 17),
(1, 0, 0, 22),
(1, 0, 1, 23),
...,
(1, 3, 2, 36)
\end{verbatim}

That is, the highest-ordered index turns fastest. — \textit{end example}]

It is possible to have degenerate generalized slices in which an address is repeated.\footnote{It is possible to have degenerate generalized slices in which an address is repeated.}

\textbf{Example 2:} If the stride parameters in the previous example are changed to \{1, 1, 1\}, the first few elements of the resulting sequence of indices will be

\begin{verbatim}
(0, 0, 0, 3),
(0, 0, 1, 4),
(0, 0, 2, 5),
(0, 1, 0, 4),
\end{verbatim}
If a degenerate slice is used as the argument to the non-const version of `operator[](const gslice&)`, the behavior is undefined.

26.7.6.2 Constructors

```cpp
gslice();
gslice(size_t start, const valarray<size_t>& lengths,
      const valarray<size_t>& strides);
```

The default constructor is equivalent to `gslice(0, valarray<size_t>(), valarray<size_t>())`. The constructor with arguments builds a `gslice` based on a specification of start, lengths, and strides, as explained in the previous subclause.

26.7.6.3 Access functions

```cpp
size_t start() const;
valarray<size_t> size() const;
valarray<size_t> stride() const;
```

1 Returns: The representation of the start, lengths, or strides specified for the `gslice`.
2 Complexity: `start()` is constant time. `size()` and `stride()` are linear in the number of strides.

26.7.7 Class template `gslice_array`

26.7.7.1 Overview

```cpp
namespace std {
  template<class T> class gslice_array {
    public:
      using value_type = T;
      void operator= (const valarray<T>&) const;
      void operator*= (const valarray<T>&) const;
      void operator/= (const valarray<T>&) const;
      void operator%= (const valarray<T>&) const;
      void operator+= (const valarray<T>&) const;
      void operator-= (const valarray<T>&) const;
      void operator^= (const valarray<T>&) const;
      void operator&= (const valarray<T>&) const;
      void operator|= (const valarray<T>&) const;
      void operator<<=(const valarray<T>&) const;
      void operator>>=(const valarray<T>&) const;
      gslice_array(const gslice_array&);
      ~gslice_array();
      const gslice_array& operator=(const gslice_array&) const;
      void operator=(const T&) const;
      gslice_array() = delete; // as implied by declaring copy constructor above
    };
  }
```

1 This template is a helper template used by the `gslice` subscript operator
2 It has reference semantics to a subset of an array specified by a `gslice` object. Thus, the expression `a[gslice(1, length, stride)] = b` has the effect of assigning the elements of `b` to a generalized slice of the elements in `a`.
26.7.7.2 Assignment

void operator=(const valarray<T>&) const;
const gslice_array& operator=(const gslice_array&) const;

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which the gslice_array refers.

26.7.7.3 Compound assignment

void operator**= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;

1 These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the gslice_array object refers.

26.7.7.4 Fill function

void operator=(const T&) const;

1 This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the gslice_array object refers.

26.7.8 Class template mask_array

26.7.8.1 Overview

namespace std {
    template<class T> class mask_array {
    public:
        using value_type = T;

        void operator**= (const valarray<T>&) const;
        void operator/= (const valarray<T>&) const;
        void operator%= (const valarray<T>&) const;
        void operator-= (const valarray<T>&) const;
        void operator+= (const valarray<T>&) const;
        void operator-= (const valarray<T>&) const;
        void operator<<=(const valarray<T>&) const;
        void operator>>=(const valarray<T>&) const;

        mask_array(const mask_array&);  
- mask_array();
        const mask_array& operator=(const mask_array&) const;
        void operator=(const T&) const;

        mask_array() = delete;   // as implied by declaring copy constructor above
    }
}

1 This template is a helper template used by the mask subscript operator:

mask_array<T> valarray<T>::operator[](const valarray<bool>&).

2 It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression a[mask] = b; has the effect of assigning the elements of b to the masked elements in a (those for which the corresponding element in mask is true).
26.7.8.2 Assignment

void operator=(const valarray<T>&) const;
const mask_array& operator=(const mask_array&) const;

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which the mask_array object refers.

26.7.8.3 Compound assignment

void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;

These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the mask_array object refers.

26.7.8.4 Fill function

void operator=(const T&) const;

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the mask_array object refers.

26.7.9 Class template indirect_array

namespace std {
    template<class T> class indirect_array {
        public:
            using value_type = T;

            void operator=(const valarray<T>&) const;
            void operator*= (const valarray<T>&) const;
            void operator/= (const valarray<T>&) const;
            void operator%= (const valarray<T>&) const;
            void operator-= (const valarray<T>&) const;
            void operator+= (const valarray<T>&) const;
            void operator-= (const valarray<T>&) const;
            void operator&= (const valarray<T>&) const;
            void operator|= (const valarray<T>&) const;
            void operator<<=(const valarray<T>&) const;
            void operator>>=(const valarray<T>&) const;

            indirect_array(const indirect_array&);
            indirect_array();
            const indirect_array& operator=(const indirect_array&); // as implied by declaring copy constructor above
            void operator=(const T&) const;

            indirect_array() = delete;
        }
    };

This template is a helper template used by the indirect subscript operator

indirect_array<T> valarray<T>::operator[](const valarray<size_t>&).

It has reference semantics to a subset of an array specified by an indirect_array. Thus, the expression a[indirect] = b; has the effect of assigning the elements of b to the elements in a whose indices appear in indirect.
26.7.9.2 Assignment

void operator=(const valarray<T>&) const;
const indirect_array& operator=(const indirect_array&) const;

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which it refers.

If the indirect_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

[Example 1:
int addr[] = {2, 3, 1, 4, 4};
valarray<size_t> indirect(addr, 5);
valarray<double> a(0., 10), b(1., 5);
a[indirect] = b;
results in undefined behavior since element 4 is specified twice in the indirection. — end example]

26.7.9.3 Compound assignment

void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;

These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the indirect_array object refers.

If the indirect_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

26.7.9.4 Fill function

void operator=(const T&) const;

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the indirect_array object refers.

26.7.10 valarray range access

In the begin and end function templates that follow, unspecified1 is a type that meets the requirements of a mutable Cpp17RandomAccessIterator (23.3.5.7) and models contiguous_iterator (23.3.4.14), whose value_type is the template parameter T and whose reference type is Tk. unspecified2 is a type that meets the requirements of a constant Cpp17RandomAccessIterator and models contiguous_iterator, whose value_type is the template parameter T and whose reference type is const Tk.

The iterators returned by begin and end for an array are guaranteed to be valid until the member function resize(size_t, T) (26.7.2.8) is called for that array or until the lifetime of that array ends, whichever happens first.

template<class T> unspecified1 begin(valarray<T>& v);
template<class T> unspecified2 begin(const valarray<T>& v);

Returns: An iterator referencing the first value in the array.

template<class T> unspecified1 end(valarray<T>& v);
template<class T> unspecified2 end(const valarray<T>& v);

Returns: An iterator referencing one past the last value in the array.
26.8 Mathematical functions for floating-point types

26.8.1 Header <cmath> synopsis

namespace std {
  using float_t = see below;
  using double_t = see below;
}

#define HUGE_VAL see below
#define HUGE_VALF see below
#define HUGE_VALL see below
#define INFINITY see below
#define NAN see below
#define FP_INFINITE see below
#define FP_NAN see below
#define FP_NORMAL see below
#define FP_SUBNORMAL see below
#define FP_ZERO see below
#define FP_FAST_FMA see below
#define FP_FAST_FMAF see below
#define FP_FAST_FMAL see below
#define FP_ILOGB0 see below
#define FP_ILOGBNAN see below
#define MATH_ERRNO see below
#define MATH_ERREXCEPT see below
#define math_errhandling see below

namespace std {
  float acos(float x); // see 16.2
double acos(double x);
long double acos(long double x); // see 16.2
float acosf(float x);
long double acosl(long double x);

float asin(float x); // see 16.2
double asin(double x);
long double asin(long double x); // see 16.2
float asinf(float x);
long double asinl(long double x);

float atan(float x); // see 16.2
double atan(double x);
long double atan(long double x); // see 16.2
float atanf(float x);
long double atanl(long double x);

float atan2(float y, float x); // see 16.2
double atan2(double y, double x);
long double atan2(long double y, long double x); // see 16.2
float atan2f(float y, float x);
long double atan2l(long double y, long double x);

float cos(float x); // see 16.2
double cos(double x);
long double cos(long double x); // see 16.2
float cosf(float x);
long double cosl(long double x);

float sin(float x); // see 16.2
double sin(double x);
long double sin(long double x); // see 16.2
float sinf(float x);
long double sinl(long double x);
float tan(float x);         // see 16.2
double tan(double x);
long double tan(long double x);  // see 16.2
float tanf(float x);
long double tanl(long double x);

float acosh(float x);        // see 16.2
double acosh(double x);
long double acosh(long double x);  // see 16.2
float acoshf(float x);
long double acoshl(long double x);

float asinh(float x);        // see 16.2
double asinh(double x);
long double asinh(long double x);  // see 16.2
float asinhf(float x);
long double asinhl(long double x);

float atanh(float x);        // see 16.2
double atanh(double x);
long double atanh(long double x);  // see 16.2
float atanhf(float x);
long double atanhl(long double x);

float cosh(float x);         // see 16.2
double cosh(double x);
long double cosh(long double x);  // see 16.2
float coshf(float x);
long double coshl(long double x);

float sinh(float x);         // see 16.2
double sinh(double x);
long double sinh(long double x);  // see 16.2
float sinhf(float x);
long double sinhl(long double x);

float tanh(float x);         // see 16.2
double tanh(double x);
long double tanh(long double x);  // see 16.2
float tanhf(float x);
long double tanhl(long double x);

float exp(float x);          // see 16.2
double exp(double x);
long double exp(long double x);  // see 16.2
float expf(float x);
long double expl(long double x);

float exp2(float x);         // see 16.2
double exp2(double x);
long double exp2(long double x);  // see 16.2
float exp2f(float x);
long double exp2l(long double x);

float expm1(float x);        // see 16.2
double expm1(double x);
long double expm1(long double x);  // see 16.2
float expmf(float x);
long double expml(long double x);

float frexp(float value, int* exp);  // see 16.2
double frexp(double value, int* exp);
long double frexp(long double value, int* exp);  // see 16.2
float frexpf(float value, int* exp);
long double frexpl(long double value, int* exp);

int ilogb(float x);  // see 16.2
int ilogb(double x);
int ilogb(long double x);  // see 16.2
int ilogbf(float x);
int ilogbl(long double x);

float ldexp(float x, int exp);  // see 16.2
double ldexp(double x, int exp);
long double ldexp(long double x, int exp);  // see 16.2
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);

float log(float x);  // see 16.2
double log(double x);
long double log(long double x);  // see 16.2
float logf(float x);
long double logl(long double x);

float log10(float x);  // see 16.2
double log10(double x);
long double log10(long double x);  // see 16.2
float log10f(float x);
long double log10l(long double x);

float log1p(float x);  // see 16.2
double log1p(double x);
long double log1p(long double x);  // see 16.2
float log1pf(float x);
long double log1pl(long double x);

float log2(float x);  // see 16.2
double log2(double x);
long double log2(long double x);  // see 16.2
float log2f(float x);
long double log2l(long double x);

float modf(float value, float* iptr);  // see 16.2
double modf(double value, double* iptr);
long double modf(long double value, long double* iptr);  // see 16.2
float modff(float value, float* iptr);
long double modfl(long double value, long double* iptr);

float scalbn(float x, int n);  // see 16.2
double scalbn(double x, int n);
long double scalbn(long double x, int n);  // see 16.2
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);

float cbrt(float x);  // see 16.2
double cbrt(double x);
long double cbrt(long double x);  // see 16.2
float cbrrt(float x);
long double cbrrl(long double x);

// 26.8.2, absolute values
int abs(int j);
long int abs(long int j);
long long int abs(long long int j);
float abs(float j);
double abs(double j);
long double abs(long double j);

float fabs(float x);  // see 16.2
double fabs(double x);
long double fabs(long double x);  // see 16.2
float fabsf(float x);
long double fabsl(long double x);

float hypot(float x, float y);  // see 16.2
double hypot(double x, double y);
long double hypot(long double x, long double y);  // see 16.2
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

// 26.8.3, three-dimensional hypotenuse
float hypot(float x, float y, float z);
double hypot(double x, double y, double z);
long double hypot(long double x, long double y, long double z);

float pow(float x, float y);  // see 16.2
double pow(double x, double y);
long double pow(long double x, long double y);  // see 16.2
float powf(float x, float y);
long double powl(long double x, long double y);

float sqrt(float x);  // see 16.2
double sqrt(double x);
long double sqrt(long double x);  // see 16.2
float sqrtf(float x);
long double sqrtl(long double x);

float erf(float x);  // see 16.2
double erf(double x);
long double erff(long double x);  // see 16.2
float erff(float x);
long double erffl(long double x);

float erfc(float x);  // see 16.2
double erfc(double x);
long double erfcf(long double x);  // see 16.2
float erfcf(float x);
long double erfcfl(long double x);

float lgamma(float x);  // see 16.2
double lgamma(double x);
long double lggamma(long double x);  // see 16.2
float lgammaf(float x);
long double lgammal(long double x);

float tgamma(float x);  // see 16.2
double tgamma(double x);
long double tgammaf(long double x);  // see 16.2
float tgammaf(float x);
long double tgammal(long double x);
float ceil(float x); // see 16.2
double ceil(double x);
long double ceil(long double x); // see 16.2
float ceilf(float x);
long double ceill(long double x);

float floor(float x); // see 16.2
double floor(double x);
long double floor(long double x); // see 16.2
float floorf(float x);
long double floorl(long double x);

float nearbyint(float x); // see 16.2
double nearbyint(double x);
long double nearbyint(long double x); // see 16.2
float nearbyintf(float x);
long double nearbyintl(long double x);

float rint(float x); // see 16.2
double rint(double x);
long double rint(long double x); // see 16.2
float rintf(float x);
long double rintl(long double x);

long int lrint(float x); // see 16.2
long int lrint(double x);
long int lrint(long double x); // see 16.2
long int lrintf(float x);
long int lrintl(long double x);

long long int llrint(float x); // see 16.2
long long int llrint(double x);
long long int llrint(long double x); // see 16.2
long long int llrintf(float x);
long long int llrintl(long double x);

float round(float x); // see 16.2
double round(double x);
long double round(long double x); // see 16.2
float roundf(float x);
long double roundl(long double x);

long int lround(float x); // see 16.2
long int lround(double x);
long int lround(long double x); // see 16.2
long int lroundf(float x);
long int lroundl(long double x);

long long int llround(float x); // see 16.2
long long int llround(double x);
long long int llround(long double x); // see 16.2
long long int llroundf(float x);
long long int llroundl(long double x);

float trunc(float x); // see 16.2
double trunc(double x);
long double trunc(long double x); // see 16.2
float truncf(float x);
long double truncl(long double x);

float fmod(float x, float y); // see 16.2
double fmod(double x, double y);
long double fmod(long double x, long double y); // see 16.2
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
float remainder(float x, float y);  // see 16.2
double remainder(double x, double y);
long double remainder(long double x, long double y);  // see 16.2
float remrnf(float x, float y);
long double remainderl(long double x, long double y);
float remquof(float x, float y, int* quo);  // see 16.2
double remquo(double x, double y, int* quo);
long double remquol(long double x, long double y, int* quo);  // see 16.2
float remquof(float x, float y, int* quo);
long double remquol(long double x, long double y, int* quo);
float copysign(float x, float y);  // see 16.2
double copysign(double x, double y);
long double copyysign(long double x, long double y);  // see 16.2
float remrnf(float x, float y);
long double remrnl(long double x, long double y);
float nanf(const char* tagp);
nanl(const char* tagp);
float nanl(const char* tagp);
float nextafterf(float x, float y);  // see 16.2
double nextafter(double x, double y);
long double nextafterl(long double x, long double y);  // see 16.2
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
float remquo(float x, float y, int* quo);  // see 16.2
double remquo(double x, double y, int* quo);
long double remquol(long double x, long double y, int* quo);  // see 16.2
float remquo(float x, float y, int* quo);
long double remquol(long double x, long double y, int* quo);
float fdim(float x, float y);  // see 16.2
double fdim(double x, double y);
long double fdiml(long double x, long double y);  // see 16.2
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
float fmax(float x, float y);  // see 16.2
double fmax(double x, double y);
long double fmaxl(long double x, long double y);  // see 16.2
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
float fmin(float x, float y);  // see 16.2
double fmin(double x, double y);
long double fminl(long double x, long double y);  // see 16.2
float fminf(float x, float y);
long double fminl(long double x, long double y);
float fma(float x, float y, float z);  // see 16.2
double fma(double x, double y, double z);
long double fmal(long double x, long double y, long double z);  // see 16.2
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

// 26.8.4, linear interpolation
constexpr float lerp(float a, float b, float t) noexcept;
constexpr double lerp(double a, double b, double t) noexcept;
constexpr long double lerp(long double a, long double b, long double t) noexcept;
// 26.8.5, classification / comparison functions
int fpclassify(float x);
int fpclassify(double x);
int fpclassify(long double x);

bool isfinite(float x);
bool isfinite(double x);
bool isfinite(long double x);

bool isinf(float x);
bool isinf(double x);
bool isinf(long double x);

bool isnan(float x);
bool isnan(double x);
bool isnan(long double x);

bool isnormal(float x);
bool isnormal(double x);
bool isnormal(long double x);

bool signbit(float x);
bool signbit(double x);
bool signbit(long double x);

bool isgreater(float x, float y);
bool isgreater(double x, double y);
bool isgreater(long double x, long double y);

bool isgreaterequal(float x, float y);
bool isgreaterequal(double x, double y);
bool isgreaterequal(long double x, long double y);

bool isless(float x, float y);
bool isless(double x, double y);
bool isless(long double x, long double y);

bool islessequal(float x, float y);
bool islessequal(double x, double y);
bool islessequal(long double x, long double y);

bool islessgreater(float x, float y);
bool islessgreater(double x, double y);
bool islessgreater(long double x, long double y);

bool isunordered(float x, float y);
bool isunordered(double x, double y);
bool isunordered(long double x, long double y);

// 26.8.6, mathematical special functions

// 26.8.6.2, associated Laguerre polynomials
double assoc_laguerre(unsigned n, unsigned m, double x);
float assoc_laguerref(unsigned n, unsigned m, float x);
long double assoc_laguerrel(unsigned n, unsigned m, long double x);

// 26.8.6.3, associated Legendre functions
double assoc_legendre(unsigned l, unsigned m, double x);
float assoc_legendref(unsigned l, unsigned m, float x);
long double assoc_legendrel(unsigned l, unsigned m, long double x);

// 26.8.6.4, beta function
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);

// 26.8.6.5, complete elliptic integral of the first kind
double comp_ellint_1(double k);
float comp_ellint_1f(float k);
long double comp_ellint_1l(long double k);

// 26.8.6.6, complete elliptic integral of the second kind
double comp_ellint_2(double k);
float comp_ellint_2f(float k);
long double comp_ellint_2l(long double k);

// 26.8.6.7, complete elliptic integral of the third kind
double comp_ellint_3(double k, double nu);
float comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);

// 26.8.6.8, regular modified cylindrical Bessel functions
double cyl_bessel_i(double nu, double x);
float cyl_bessel_if(float nu, float x);
long double cyl_bessel_il(long double nu, long double x);

// 26.8.6.9, cylindrical Bessel functions of the first kind
double cyl_bessel_j(double nu, double x);
float cyl_bessel_jf(float nu, float x);
long double cyl_bessel_jl(long double nu, long double x);

// 26.8.6.10, irregular modified cylindrical Bessel functions
double cyl_bessel_k(double nu, double x);
float cyl_bessel_kf(float nu, float x);
long double cyl_bessel_kl(long double nu, long double x);

// 26.8.6.11, cylindrical Neumann functions;
// cylindrical Bessel functions of the second kind
double cyl_neumann(double nu, double x);
float cyl_neumannf(float nu, float x);
long double cyl_neumannl(long double nu, long double x);

// 26.8.6.12, incomplete elliptic integral of the first kind
double ellint_1(double k, double phi);
float ellint_1f(float k, float phi);
long double ellint_1l(long double k, long double phi);

// 26.8.6.13, incomplete elliptic integral of the second kind
double ellint_2(double k, double phi);
float ellint_2f(float k, float phi);
long double ellint_2l(long double k, long double phi);

// 26.8.6.14, incomplete elliptic integral of the third kind
double ellint_3(double k, double nu, double phi);
float ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu, long double phi);

// 26.8.6.15, exponential integral
double expint(double x);
float expintf(float x);
long double expintl(long double x);

// 26.8.6.16, Hermite polynomials
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);
// 26.8.6.17, Laguerre polynomials
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

// 26.8.6.18, Legendre polynomials
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);

// 26.8.6.19, Riemann zeta function
double riemann_zeta(double x);
float riemann_zetaf(float x);
long double riemann_zetal(long double x);

// 26.8.6.20, spherical Bessel functions of the first kind
double sph_bessel(unsigned n, double x);
float sph_besself(unsigned n, float x);
long double sph_bessell(unsigned n, long double x);

// 26.8.6.21, spherical associated Legendre functions
double sph_legendre(unsigned l, unsigned m, double theta);
float sph_legendref(unsigned l, unsigned m, float theta);
long double sph_legendrel(unsigned l, unsigned m, long double theta);

// 26.8.6.22, spherical Neumann functions;
// spherical Bessel functions of the second kind
double sph_neumann(unsigned n, double x);
float sph_neumannf(unsigned n, float x);
long double sph_neumannl(unsigned n, long double x);

}  

1 The contents and meaning of the header `<cmath>` are the same as the C standard library header `<math.h>`, with the addition of a three-dimensional hypotenuse function (26.8.3) and the mathematical special functions described in 26.8.6.

[Note 1: Several functions have additional overloads in this document, but they have the same behavior as in the C standard library (16.2). — end note]

2 For each set of overloaded functions within `<cmath>`, with the exception of `abs`, there shall be additional overloads sufficient to ensure:

(2.1) — If any argument of arithmetic type corresponding to a `double` parameter has type `long double`, then all arguments of arithmetic type (6.8.2) corresponding to `double` parameters are effectively cast to `long double`.

(2.2) — Otherwise, if any argument of arithmetic type corresponding to a `double` parameter has type `double` or an integer type, then all arguments of arithmetic type corresponding to `double` parameters are effectively cast to `double`.

(2.3) — [Note 2: Otherwise, all arguments of arithmetic type corresponding to `double` parameters have type `float`. — end note]

[Note 3: `abs` is exempted from these rules in order to stay compatible with C. — end note]

See also: ISO C 7.12

26.8.2 Absolute values [c.math.abs]

1 [Note 1: The headers `<cstdlib>` (17.2.2) and `<cmath>` (26.8.1) declare the functions described in this subclause. — end note]

int abs(int j);
long int abs(long int j);
long long int abs(long long int j);
float abs(float j);
double abs(double j);
long double abs(long double j);

Effects: The abs functions have the semantics specified in the C standard library for the functions abs, labs, llabs, fabsf, fabs, and fabsl.

Remarks: If abs() is called with an argument of type X for which is_unsigned_v<X> is true and if X cannot be converted to int by integral promotion (7.3.7), the program is ill-formed.

[Note 2: Arguments that can be promoted to int are permitted for compatibility with C. — end note]

See also: ISO C 7.12.7.2, 7.22.6.1

26.8.3 Three-dimensional hypotenuse

float hypot(float x, float y, float z);
double hypot(double x, double y, double z);
long double hypot(long double x, long double y, long double z);

Returns: \( \sqrt{x^2 + y^2 + z^2} \).

26.8.4 Linear interpolation

constexpr float lerp(float a, float b, float t) noexcept;
customexpr double lerp(double a, double b, double t) noexcept;
customexpr long double lerp(long double a, long double b, long double t) noexcept;

Returns: \( a + t(b - a) \).

Remarks: Let \( r \) be the value returned. If isfinite(a) && isfinite(b), then:

- If \( t == 0 \), then \( r == a \).
- If \( t == 1 \), then \( r == b \).
- If \( t >= 0 && t <= 1 \), then isfinite(r).
- If isfinite(t) && a == b, then \( r == a \).
- If isfinite(t) || !isnan(t) && b-a != 0, then !isnan(r).

Let CMP(x,y) be 1 if \( x > y \), -1 if \( x < y \), and 0 otherwise. For any \( t_1 \) and \( t_2 \), the product of CMP(lerp(a, b, t2), lerp(a, b, t1)), CMP(t2, t1), and CMP(b, a) is non-negative.

26.8.5 Classification / comparison functions

The classification / comparison functions behave the same as the C macros with the corresponding names defined in the C standard library. Each function is overloaded for the three floating-point types.

See also: ISO C 7.12.3, 7.12.4

26.8.6 Mathematical special functions

26.8.6.1 General

If any argument value to any of the functions specified in 26.8.6 is a NaN (Not a Number), the function shall return a NaN but it shall not report a domain error. Otherwise, the function shall report a domain error for just those argument values for which:

- the function description’s Returns: element explicitly specifies a domain and those argument values fall outside the specified domain, or
- the corresponding mathematical function value has a nonzero imaginary component, or
- the corresponding mathematical function is not mathematically defined.260

Unless otherwise specified, each function is defined for all finite values, for negative infinity, and for positive infinity.

26.8.6.2 Associated Laguerre polynomials

double assoc_laguerre(unsigned n, unsigned m, double x);
float assoc_laguerref(unsigned n, unsigned m, float x);

260) A mathematical function is mathematically defined for a given set of argument values (a) if it is explicitly defined for that set of argument values, or (b) if its limiting value exists and does not depend on the direction of approach.
long double assoc_laguerrel(unsigned n, unsigned m, long double x);

Effects: These functions compute the associated Laguerre polynomials of their respective arguments \( n \), \( m \), and \( x \).

Returns:
\[
L_n^m(x) = (-1)^m \frac{d^m}{dx^m} L_{n+m}(x), \quad \text{for } x \geq 0,
\]
where \( n \) is \( n \), \( m \) is \( m \), and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( n \geq 128 \) or if \( m \geq 128 \).

26.8.6.3 Associated Legendre functions

double assoc_legendre(unsigned l, unsigned m, double x);
float assoc_legendref(unsigned l, unsigned m, float x);
long double assoc_legendrel(unsigned l, unsigned m, long double x);

Effects: These functions compute the associated Legendre functions of their respective arguments \( l \), \( m \), and \( x \).

Returns:
\[
P_m^l(x) = (1 - x^2)^{m/2} \frac{d^m}{dx^m} P_l(x), \quad \text{for } |x| \leq 1,
\]
where \( l \) is \( l \), \( m \) is \( m \), and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( l \geq 128 \).

26.8.6.4 Beta function

double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);

Effects: These functions compute the beta function of their respective arguments \( x \) and \( y \).

Returns:
\[
B(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}, \quad \text{for } x > 0, \ y > 0,
\]
where \( x \) is \( x \) and \( y \) is \( y \).

26.8.6.5 Complete elliptic integral of the first kind

double comp_ellint_1(double k);
float comp_ellint_1f(float k);
long double comp_ellint_1l(long double k);

Effects: These functions compute the complete elliptic integral of the first kind of their respective arguments \( k \).

Returns:
\[
K(k) = F(k,\pi/2), \quad \text{for } |k| \leq 1,
\]
where \( k \) is \( k \).

See also 26.8.6.12.

26.8.6.6 Complete elliptic integral of the second kind

double comp_ellint_2(double k);
float comp_ellint_2f(float k);
long double comp_ellint_2l(long double k);

Effects: These functions compute the complete elliptic integral of the second kind of their respective arguments \( k \).

Returns:
\[
E(k) = E(k,\pi/2), \quad \text{for } |k| \leq 1,
\]
where \( k \) is \( k \).

See also 26.8.6.13.
26.8.6.7 Complete elliptic integral of the third kind

\begin{verbatim}
double comp_ellint_3(double k, double nu);
float comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);
\end{verbatim}

1 \textit{Effects}: These functions compute the complete elliptic integral of the third kind of their respective arguments \(k\) and \(\nu\).

2 \textit{Returns}:
\[ \Pi(\nu, k) = \Pi(\nu, k, \pi/2), \quad \text{for } |k| \leq 1, \]
where \(k\) is \(k\) and \(\nu\) is \(\nu\).

3 See also 26.8.6.14.

26.8.6.8 Regular modified cylindrical Bessel functions

\begin{verbatim}
double cyl_bessel_i(double nu, double x);
float cyl_bessel_if(float nu, float x);
long double cyl_bessel_il(long double nu, long double x);
\end{verbatim}

1 \textit{Effects}: These functions compute the regular modified cylindrical Bessel functions of their respective arguments \(\nu\) and \(x\).

2 \textit{Returns}:
\[ I_\nu(x) = \sum_{k=0}^{\infty} \frac{(x/2)^{\nu+2k}}{k! \Gamma(\nu + k + 1)}, \quad \text{for } x \geq 0, \]
where \(\nu\) is \(\nu\) and \(x\) is \(x\).

3 \textit{Remarks}: The effect of calling each of these functions is implementation-defined if \(\nu \geq 128\).

4 See also 26.8.6.9.

26.8.6.9 Cylindrical Bessel functions of the first kind

\begin{verbatim}
double cyl_bessel_j(double nu, double x);
float cyl_bessel_jf(float nu, float x);
long double cyl_bessel_jl(long double nu, long double x);
\end{verbatim}

1 \textit{Effects}: These functions compute the cylindrical Bessel functions of the first kind of their respective arguments \(\nu\) and \(x\).

2 \textit{Returns}:
\[ J_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{\nu+2k}}{k! \Gamma(\nu + k + 1)}, \quad \text{for } x \geq 0, \]
where \(\nu\) is \(\nu\) and \(x\) is \(x\).

3 \textit{Remarks}: The effect of calling each of these functions is implementation-defined if \(\nu \geq 128\).

4 See also 26.8.6.8, 26.8.6.9, 26.8.6.10.

26.8.6.10 Irregular modified cylindrical Bessel functions

\begin{verbatim}
double cyl_bessel_k(double nu, double x);
float cyl_bessel_kf(float nu, float x);
long double cyl_bessel_kl(long double nu, long double x);
\end{verbatim}

1 \textit{Effects}: These functions compute the irregular modified cylindrical Bessel functions of their respective arguments \(\nu\) and \(x\).

2 \textit{Returns}:
\[ K_\nu(x) = \frac{\pi}{2} \left( \frac{1 - J_\nu(ix)}{\sin \nu \pi} - \frac{1 - J_\nu(x)}{\sin \nu \pi} \right), \quad \text{for } x \geq 0 \text{ and non-integral } \nu \]
\[ = \lim_{\mu \to \nu} \frac{1 - J_\nu(x)}{\sin \mu \pi}, \quad \text{for } x \geq 0 \text{ and integral } \nu \]
where \(\nu\) is \(\nu\) and \(x\) is \(x\).

3 \textit{Remarks}: The effect of calling each of these functions is implementation-defined if \(\nu \geq 128\).

4 See also 26.8.6.8, 26.8.6.9, 26.8.6.11.
26.8.6.11 Cylindrical Neumann functions

cyl_neumann(double nu, double x);
cyl_neumannf(float nu, float x);
cyl_neumannl(long double nu, long double x);

Effects: These functions compute the cylindrical Neumann functions, also known as the cylindrical Bessel functions of the second kind, of their respective arguments nu and x.

Returns:
\[ N_\nu(x) = \begin{cases} 
\frac{J_\nu(x) \cos \nu \pi - J_{-\nu}(x)}{\sin \nu \pi}, & \text{for } x \geq 0 \text{ and non-integral } \nu \\
\lim_{\mu \to \nu} \frac{J_\mu(x) \cos \mu \pi - J_{-\mu}(x)}{\sin \mu \pi}, & \text{for } x \geq 0 \text{ and integral } \nu 
\end{cases} \]

where \( \nu \) is nu and \( x \) is x.

Remarks: The effect of calling each of these functions is implementation-defined if \( \nu \geq 128 \).

See also 26.8.6.9.

26.8.6.12 Incomplete elliptic integral of the first kind

eellint_1(double k, double phi);
eellint_1f(float k, float phi);
eellint_1l(long double k, long double phi);

Effects: These functions compute the incomplete elliptic integral of the first kind of their respective arguments k and phi (phi measured in radians).

Returns:
\[ F(k, \phi) = \int_0^\phi \frac{d\theta}{\sqrt{1 - k^2 \sin^2 \theta}}, \quad \text{for } |k| \leq 1, \]

where \( k \) is k and \( \phi \) is \( \phi \).

26.8.6.13 Incomplete elliptic integral of the second kind

eellint_2(double k, double phi);
eellint_2f(float k, float phi);
eellint_2l(long double k, long double phi);

Effects: These functions compute the incomplete elliptic integral of the second kind of their respective arguments k and phi (phi measured in radians).

Returns:
\[ E(k, \phi) = \int_0^\phi \sqrt{1 - k^2 \sin^2 \theta} d\theta, \quad \text{for } |k| \leq 1, \]

where \( k \) is k and \( \phi \) is \( \phi \).

26.8.6.14 Incomplete elliptic integral of the third kind

eellint_3(double k, double nu, double phi);
eellint_3f(float k, float nu, float phi);
eellint_3l(long double k, long double nu, long double phi);

Effects: These functions compute the incomplete elliptic integral of the third kind of their respective arguments k, nu, and phi (phi measured in radians).

Returns:
\[ \Pi(\nu, k, \phi) = \int_0^\phi \frac{d\theta}{(1 - \nu \sin^2 \theta)\sqrt{1 - k^2 \sin^2 \theta}}, \quad \text{for } |k| \leq 1, \]

where \( \nu \) is nu, \( k \) is k, and \( \phi \) is \( \phi \).
26.8.6.15  Exponential integral

double expint(double x);
float expintf(float x);
long double expintl(long double x);

Effects: These functions compute the exponential integral of their respective arguments \( x \).

Returns:

\[
Ei(x) = - \int_{-x}^{\infty} \frac{e^{-t}}{t} \, dt
\]

where \( x \) is \( x \).

26.8.6.16  Hermite polynomials

double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);

Effects: These functions compute the Hermite polynomials of their respective arguments \( n \) and \( x \).

Returns:

\[
H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}
\]

where \( n \) is \( n \) and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).

26.8.6.17  Laguerre polynomials

double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

Effects: These functions compute the Laguerre polynomials of their respective arguments \( n \) and \( x \).

Returns:

\[
L_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n} \left( x^n e^{-x} \right), \text{ for } x \geq 0,
\]

where \( n \) is \( n \) and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).

26.8.6.18  Legendre polynomials

double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);

Effects: These functions compute the Legendre polynomials of their respective arguments \( l \) and \( x \).

Returns:

\[
P_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} \left( x^2 - 1 \right)^l, \text{ for } |x| \leq 1,
\]

where \( l \) is \( l \) and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( l \geq 128 \).

26.8.6.19  Riemann zeta function

double riemann_zeta(double x);
float riemann_zetaf(float x);
long double riemann_zetal(long double x);

Effects: These functions compute the Riemann zeta function of their respective arguments \( x \).
\begin{align*}
\zeta(x) &= \begin{cases}
\sum_{k=1}^{\infty} k^{-x}, & \text{for } x > 1 \\
\frac{1}{1 - 2^{1-x}} \sum_{k=1}^{\infty} (-1)^{k-1} k^{-x}, & \text{for } 0 \leq x \leq 1 \\
2^{\frac{1}{2}} \pi^{x-1} \sin\left(\frac{\pi x}{2}\right) \Gamma(1-x) \zeta(1-x), & \text{for } x < 0
\end{cases}
\end{align*}

where \( x \) is \( x \).

\subsection*{26.8.6.20 Spherical Bessel functions of the first kind} 
\[ \text{sf.cmath.sph.bessel} \]
\begin{verbatim}
double sph_bessel(unsigned n, double x);
float sph_besself(unsigned n, float x);
long double sph_bessell(unsigned n, long double x);
\end{verbatim}

\begin{enumerate}
\item \textit{Effects:} These functions compute the spherical Bessel functions of the first kind of their respective arguments \( n \) and \( x \).
\item \textit{Returns:} 
\[ j_n(x) = (\pi/2x)^{1/2} J_{n+1/2}(x), \text{ for } x \geq 0, \]
where \( n \) is \( n \) and \( x \) is \( x \).
\item \textit{Remarks:} The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).
\item See also \ref{26.8.6.9}.
\end{enumerate}

\subsection*{26.8.6.21 Spherical associated Legendre functions} 
\[ \text{sf.cmath.sph.legendre} \]
\begin{verbatim}
double sph_legendre(unsigned l, unsigned m, double theta);
float sph_legendref(unsigned l, unsigned m, float theta);
long double sph_legendrel(unsigned l, unsigned m, long double theta);
\end{verbatim}

\begin{enumerate}
\item \textit{Effects:} These functions compute the spherical associated Legendre functions of their respective arguments \( l, m \), and \( \theta \) (\( \theta \) measured in radians).
\item \textit{Returns:} 
\[ Y^m_\ell(\theta, 0) \]
where
\[ Y^m_\ell(\theta, \phi) = (-1)^m \left[ \frac{(2\ell + 1)(\ell - m)!}{4\pi (\ell + m)!} \right]^{1/2} P^m_\ell(\cos \theta) e^{im\phi}, \text{ for } |m| \leq \ell, \]
and \( l \) is \( l \), \( m \) is \( m \), and \( \theta \) is \( \theta \).
\item \textit{Remarks:} The effect of calling each of these functions is implementation-defined if \( l \geq 128 \).
\item See also \ref{26.8.6.3}.
\end{enumerate}

\subsection*{26.8.6.22 Spherical Neumann functions} 
\[ \text{sf.cmath.sph.neumann} \]
\begin{verbatim}
double sph_neumann(unsigned n, double x);
float sph_neumannf(unsigned n, float x);
long double sph_neumannl(unsigned n, long double x);
\end{verbatim}

\begin{enumerate}
\item \textit{Effects:} These functions compute the spherical Neumann functions, also known as the spherical Bessel functions of the second kind, of their respective arguments \( n \) and \( x \).
\item \textit{Returns:} 
\[ n_n(x) = (\pi/2x)^{1/2} N_{n+1/2}(x), \text{ for } x \geq 0, \]
where \( n \) is \( n \) and \( x \) is \( x \).
\item \textit{Remarks:} The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).
\item See also \ref{26.8.6.11}.
\end{enumerate}
namespace std::numbers {
  template<class T> inline constexpr T e_v = unspecified;
  template<class T> inline constexpr T log2e_v = unspecified;
  template<class T> inline constexpr T log10e_v = unspecified;
  template<class T> inline constexpr T pi_v = unspecified;
  template<class T> inline constexpr T inv_pi_v = unspecified;
  template<class T> inline constexpr T inv_sqrtpi_v = unspecified;
  template<class T> inline constexpr T ln2_v = unspecified;
  template<class T> inline constexpr T ln10_v = unspecified;
  template<class T> inline constexpr T sqrt2_v = unspecified;
  template<class T> inline constexpr T sqrt3_v = unspecified;
  template<class T> inline constexpr T inv_sqrt3_v = unspecified;
  template<class T> inline constexpr T egamma_v = unspecified;
  template<class T> inline constexpr T phi_v = unspecified;
}

namespace 

template<floating_point T> inline constexpr T e_v<T> = see below;
template<floating_point T> inline constexpr T log2e_v<T> = see below;
template<floating_point T> inline constexpr T log10e_v<T> = see below;
template<floating_point T> inline constexpr T pi_v<T> = see below;
template<floating_point T> inline constexpr T inv_pi_v<T> = see below;
template<floating_point T> inline constexpr T inv_sqrtpi_v<T> = see below;
template<floating_point T> inline constexpr T ln2_v<T> = see below;
template<floating_point T> inline constexpr T ln10_v<T> = see below;
template<floating_point T> inline constexpr T sqrt2_v<T> = see below;
template<floating_point T> inline constexpr T sqrt3_v<T> = see below;
template<floating_point T> inline constexpr T inv_sqrt3_v<T> = see below;
template<floating_point T> inline constexpr T egamma_v<T> = see below;
template<floating_point T> inline constexpr T phi_v<T> = see below;

inline constexpr double e = e_v<double>;
inline constexpr double log2e = log2e_v<double>;
inline constexpr double log10e = log10e_v<double>;
inline constexpr double pi = pi_v<double>;
inline constexpr double inv_pi = inv_pi_v<double>;
inline constexpr double inv_sqrtpi = inv_sqrtpi_v<double>;
inline constexpr double ln2 = ln2_v<double>;
inline constexpr double ln10 = ln10_v<double>;
inline constexpr double sqrt2 = sqrt2_v<double>;
inline constexpr double sqrt3 = sqrt3_v<double>;
inline constexpr double inv_sqrt3 = inv_sqrt3_v<double>;
inline constexpr double egamma = egamma_v<double>;
inline constexpr double phi = phi_v<double>;
}
27 Time library

27.1 General

This Clause describes the chrono library (27.2) and various C functions (27.14) that provide generally useful time utilities, as summarized in Table 96.

Table 96: Time library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.3</td>
<td>Cpp17Clock requirements</td>
</tr>
<tr>
<td>27.4</td>
<td>Time-related traits</td>
</tr>
<tr>
<td>27.5</td>
<td>Class template duration</td>
</tr>
<tr>
<td>27.6</td>
<td>Class template time_point</td>
</tr>
<tr>
<td>27.7</td>
<td>Clocks</td>
</tr>
<tr>
<td>27.8</td>
<td>Civil calendar</td>
</tr>
<tr>
<td>27.9</td>
<td>Class template hh_mm_ss</td>
</tr>
<tr>
<td>27.10</td>
<td>12/24 hour functions</td>
</tr>
<tr>
<td>27.11</td>
<td>Time zones</td>
</tr>
<tr>
<td>27.12</td>
<td>Formatting</td>
</tr>
<tr>
<td>27.13</td>
<td>Parsing</td>
</tr>
<tr>
<td>27.14</td>
<td>C library time utilities</td>
</tr>
</tbody>
</table>

2 Let STATICALLY-WIDEN<charT>("...") be ",..." if charT is char and L"..." if charT is wchar_t.

27.2 Header <chrono> synopsis

#include <compare>    // see 17.11.1

namespace std {
    namespace chrono {
        // 27.5, class template duration
        template<class Rep, class Period = ratio<1>> class duration;

        // 27.6, class template time_point
        template<class Clock, class Duration = typename Clock::duration> class time_point;
    }

    // 27.4.3, common_type specializations
    template<class Rep1, class Period1, class Rep2, class Period2>
    struct common_type<chrono::duration<Rep1, Period1>,
                       chrono::duration<Rep2, Period2>>;

    template<class Clock, class Duration1, class Duration2>
    struct common_type<chrono::time_point<Clock, Duration1>,
                       chrono::time_point<Clock, Duration2>>;

    namespace chrono {
        // 27.4, customization traits
        template<class Rep> struct treat_as_floating_point;
        template<class Rep>
            inline constexpr bool treat_as_floating_point_v = treat_as_floating_point<Rep>::value;

        template<class Rep> struct duration_values;

        template<class T> struct is_clock;
        template<class T> inline constexpr bool is_clock_v = is_clock<T>::value;

§ 27.2

1245
// 27.5.6, duration arithmetic

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

```cpp
template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator*(const duration<Rep1, Period>& d, const Rep2& s);
```

```cpp
template<class Rep1, class Rep2, class Period>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator*(const Rep1& s, const duration<Rep2, Period>& d);
```

```cpp
template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator/(const duration<Rep1, Period>& d, const Rep2& s);
```

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr common_type_t<Rep1, Rep2>
operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

```cpp
template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator%(const duration<Rep1, Period>& d, const Rep2& s);
```

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

// 27.5.7, duration comparisons

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);
```

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);
```

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);
```

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);
```

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);
```

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
requires see below
constexpr auto operator<=>(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);
```

// 27.5.8, conversions

```cpp
template<class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
```

```cpp
template<class ToDuration, class Rep, class Period>
constexpr ToDuration floor(const duration<Rep, Period>& d);
```

```cpp
template<class ToDuration, class Rep, class Period>
constexpr ToDuration ceil(const duration<Rep, Period>& d);
```

```cpp
template<class ToDuration, class Rep, class Period>
constexpr ToDuration round(const duration<Rep, Period>& d);
```

// 27.5.11, duration I/O

```cpp
template<class charT, class traits, class Rep, class Period>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
const duration<Rep, Period>& d);
```
template<class charT, class traits, class Rep, class Period, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                duration<Rep, Period>& d,
                basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

// convenience typedefs
using nanoseconds = duration<
    signed integer type of at least 64 bits, nano>;
using microseconds = duration<
    signed integer type of at least 55 bits, micro>;
using milliseconds = duration<
    signed integer type of at least 45 bits, milli>;
using seconds = duration<
    signed integer type of at least 35 bits>;
using minutes = duration<
    signed integer type of at least 29 bits, ratio< 60>>;
using hours = duration<
    signed integer type of at least 23 bits, ratio<3600>>;
using days = duration<
    signed integer type of at least 25 bits,
    ratio_multiply<ratio<24>, hours::period>>;
using weeks = duration<
    signed integer type of at least 22 bits,
    ratio_multiply<ratio<7>, days::period>>;
using years = duration<
    signed integer type of at least 17 bits,
    ratio_multiply<ratio<146097, 400>, days::period>>;
using months = duration<
    signed integer type of at least 20 bits,
    ratio_divide<years::period, ratio<12>>>

// 27.6.6, time_point arithmetic
template<class Clock, class Duration1, class Rep2, class Period2>
    constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
        operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
template<class Rep1, class Period1, class Clock, class Duration2>
    constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>>
        operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);
template<class Clock, class Duration1, class Rep2, class Period2>
    constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>
        operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
template<class Clock, class Duration1, class Duration2>
    constexpr common_type_t<Duration1, Duration2>
        operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// 27.6.7, time_point comparisons
template<class Clock, class Duration1, class Duration2>
    constexpr bool operator==(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template<class Clock, class Duration1, class Duration2>
    constexpr bool operator<(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template<class Clock, class Duration1, class Duration2>
    constexpr bool operator>(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template<class Clock, class Duration1, class Duration2>
    constexpr bool operator<=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template<class Clock, class Duration1, class Duration2>
    constexpr bool operator>=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

    constexpr auto operator<=>(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// 27.6.8, conversions
template<class ToDuration, class Clock, class Duration>
    constexpr time_point<Clock, ToDuration>
        time_point_cast(const time_point<Clock, Duration>& t);
template<class ToDuration, class Clock, class Duration>
    constexpr time_point<Clock, ToDuration>
        floor(const time_point<Clock, Duration>& tp);
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> ceil(const time_point<Clock, Duration>& tp);

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> round(const time_point<Clock, Duration>& tp);

// 27.5.10, specialized algorithms
template<class Rep, class Period>
constexpr duration<Rep, Period> abs(duration<Rep, Period> d);

// 27.7.2, class system_clock
class system_clock;

template<class Duration>
using sys_time = time_point<system_clock, Duration>;

using sys_seconds = sys_time<seconds>;
using sys_days = sys_time<days>;

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const sys_time<Duration>& tp);

template<class charT, class traits>  
operator<<(basic_ostream<charT, traits>& os, const sys_days& dp);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,  
    sys_time<Duration>& tp,  
    basic_string<charT, traits, Alloc>* abbrev = nullptr,  
    minutes* offset = nullptr);

// 27.7.3, class utc_clock
class utc_clock;

template<class Duration>
using utc_time = time_point<utc_clock, Duration>;

using utc_seconds = utc_time<seconds>;

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const utc_time<Duration>& t);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,  
    utc_time<Duration>& tp,  
    basic_string<charT, traits, Alloc>* abbrev = nullptr,  
    minutes* offset = nullptr);

struct leap_second_info;

template<class Duration>
leap_second_info get_leap_second_info(const utc_time<Duration>& ut);

// 27.7.4, class tai_clock
class tai_clock;

template<class Duration>
using tai_time = time_point<tai_clock, Duration>;

using tai_seconds = tai_time<seconds>;

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const tai_time<Duration>& t);
template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
    tai_time<Duration>& tp,
    basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 27.7.5, class gps_clock
class gps_clock;

#include<chrono>

#include<ctime>

using gps_time = time_point<gps_clock, Duration>;
using gps_seconds = gps_time<seconds>;

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const gps_time<Duration>& t);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
    gps_time<Duration>& tp,
    basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 27.7.6, type file_clock
using file_clock = see below;

template<class Duration>
    using file_time = time_point<file_clock, Duration>;

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const file_time<Duration>& t);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
    file_time<Duration>& tp,
    basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 27.7.7, class steady_clock
class steady_clock;

// 27.7.8, class high_resolution_clock
class high_resolution_clock;

// 27.7.9, local time
struct local_t {};

#include<chrono>

template<class Duration>
    using local_time = time_point<local_t, Duration>;

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const local_time<Duration>& t);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
    local_time<Duration>& tp,
    basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);
// 27.7.10, time_point conversions
template<class DestClock, class SourceClock>
    struct clock_time_conversion;

template<class DestClock, class SourceClock, class Duration>
    auto clock_cast(const time_point<SourceClock, Duration>& t);

// 27.8.2, class last_spec
struct last_spec;

// 27.8.3, class day
class day;

constexpr bool operator==(const day& x, const day& y) noexcept;
constexpr strong_ordering operator<=>(const day& x, const day& y) noexcept;

constexpr day operator+(const day& x, const days& y) noexcept;
constexpr day operator+(const days& x, const day& y) noexcept;
constexpr day operator-(const day& x, const days& y) noexcept;
constexpr days operator-(const day& x, const day& y) noexcept;

template<class charT, class traits>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const day& d);

template<class charT, class traits, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
        from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                     day& d, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                     minutes* offset = nullptr);

// 27.8.4, class month
class month;

constexpr bool operator==(const month& x, const month& y) noexcept;
constexpr strong_ordering operator<=>(const month& x, const month& y) noexcept;

constexpr month operator+(const month& x, const months& y) noexcept;
constexpr month operator+(const months& x, const month& y) noexcept;
constexpr month operator-(const month& x, const months& y) noexcept;
constexpr months operator-(const month& x, const month& y) noexcept;

template<class charT, class traits>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const month& m);

template<class charT, class traits, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
        from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                     month& m, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                     minutes* offset = nullptr);

// 27.8.5, class year
class year;

constexpr bool operator==(const year& x, const year& y) noexcept;
constexpr strong_ordering operator<=>(const year& x, const year& y) noexcept;

constexpr year operator+(const year& x, const years& y) noexcept;
constexpr year operator+(const years& x, const year& y) noexcept;
constexpr year operator-(const year& x, const years& y) noexcept;
constexpr years operator-(const year& x, const year& y) noexcept;

template<class charT, class traits>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const year& y);
template<class charT, class traits, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                year& y, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

// 27.8.6, class weekday
class weekday;

constexpr bool operator==(const weekday& x, const weekday& y) noexcept;
constexpr weekday operator+(const weekday& x, const days& y) noexcept;
constexpr weekday operator+(const days& x, const weekday& y) noexcept;
constexpr weekday operator-(const weekday& x, const days& y) noexcept;
constexpr days operator-(const weekday& x, const weekday& y) noexcept;

template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const weekday& wd);

template<class charT, class traits, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                weekday& wd, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

// 27.8.7, class weekday_indexed
class weekday_indexed;

constexpr bool operator==(const weekday_indexed& x, const weekday_indexed& y) noexcept;

template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const weekday_indexed& wdi);

// 27.8.8, class weekday_last
class weekday_last;

constexpr bool operator==(const weekday_last& x, const weekday_last& y) noexcept;

template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const weekday_last& wdl);

// 27.8.9, class month_day
class month_day;

constexpr bool operator==(const month_day& x, const month_day& y) noexcept;
constexpr strong_ordering operator<=>(const month_day& x, const month_day& y) noexcept;

template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const month_day& md);

template<class charT, class traits, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                month_day& md, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

// 27.8.10, class month_day_last
class month_day_last;

§ 27.2 1251
constexpr bool operator==(const month_day_last& x, const month_day_last& y) noexcept;
constexpr strong_ordering operator<=>(const month_day_last& x,
const month_day_last& y) noexcept;

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_day_last& mdl);

// 27.8.11, class month_weekday
class month_weekday;
constexpr bool operator==(const month_weekday& x, const month_weekday& y) noexcept;

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_weekday& mwd);

// 27.8.12, class month_weekday_last
class month_weekday_last;
constexpr bool operator==(const month_weekday_last& x, const month_weekday_last& y) noexcept;

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_weekday_last& mwdl);

// 27.8.13, class year_month
class year_month;
constexpr bool operator==(const year_month& x, const year_month& y) noexcept;
constexpr strong_ordering operator<=>(const year_month& x,
const year_month& y) noexcept;

customexpr year_month operator+(const year_month& ym, const months& dm) noexcept;
customexpr year_month operator+(const months& dm, const year_month& ym) noexcept;
customexpr year_month operator-(const year_month& ym, const months& dm) noexcept;
customexpr months operator-(const year_month& x, const year_month& y) noexcept;
customexpr year_month operator+(const years& dy, const year_month& ym) noexcept;
customexpr year_month operator+(const year_month& ym, const years& dy) noexcept;

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month& ym);

// 27.8.14, class year_month_day
class year_month_day;
constexpr bool operator==(const year_month_day& x, const year_month_day& y) noexcept;
constexpr strong_ordering operator<=>(const year_month_day& x,
const year_month_day& y) noexcept;

customexpr year_month_day operator+(const year_month_day& ymd, const months& dm) noexcept;
customexpr year_month_day operator+(const months& dm, const year_month_day& ymd) noexcept;
customexpr year_month_day operator+(const years& dy, const year_month_day& ymd) noexcept;
customexpr year_month_day operator-(const year_month_day& ymd, const months& dm) noexcept;
customexpr year_month_day operator-(const years& dy, const year_month_day& ymd) noexcept;

§ 27.2
template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const year_month_day& ymd);

template<class charT, class traits, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                year_month_day& ymd, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

// 27.8.15, class year_month_day_last
class year_month_day_last;

constexpr bool operator==(const year_month_day_last& x,
                          const year_month_day_last& y) noexcept;
constexpr strong_ordering operator<=>(const year_month_day_last& x,
                                       const year_month_day_last& y) noexcept;

constexpr year_month_day_last
    operator+(const year_month_day_last& ymdl, const months& dm) noexcept;
constexpr year_month_day_last
    operator+(const months& dm, const year_month_day_last& ymdl) noexcept;
constexpr year_month_day_last
    operator+(const year_month_day_last& ymdl, const years& dy) noexcept;
constexpr year_month_day_last
    operator+(const years& dy, const year_month_day_last& ymdl) noexcept;
constexpr year_month_day_last
    operator-(const year_month_day_last& ymdl, const months& dm) noexcept;
constexpr year_month_day_last
    operator-(const year_month_day_last& ymdl, const years& dy) noexcept;

template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const year_month_day_last& ymdl);

// 27.8.16, class year_month_weekday
class year_month_weekday;

constexpr bool operator==(const year_month_weekday& x,
                          const year_month_weekday& y) noexcept;

constexpr year_month_weekday
    operator+(const year_month_weekday& ymwd, const months& dm) noexcept;
constexpr year_month_weekday
    operator+(const months& dm, const year_month_weekday& ymwd) noexcept;
constexpr year_month_weekday
    operator+(const year_month_weekday& ymwd, const years& dy) noexcept;
constexpr year_month_weekday
    operator+(const years& dy, const year_month_weekday& ymwd) noexcept;
constexpr year_month_weekday
    operator-(const year_month_weekday& ymwd, const months& dm) noexcept;
constexpr year_month_weekday
    operator-(const year_month_weekday& ymwd, const years& dy) noexcept;

template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const year_month_weekday& ymwd);

// 27.8.17, class year_month_weekday_last
class year_month_weekday_last;

constexpr bool operator==(const year_month_weekday_last& x,
                          const year_month_weekday_last& y) noexcept;
constexpr year_month_weekday_last
  operator+(const year_month_weekday_last& ymwdl, const months& dm) noexcept;
constexpr year_month_weekday_last
  operator+(const months& dm, const year_month_weekday_last& ymwdl) noexcept;
constexpr year_month_weekday_last
  operator+(const year_month_weekday_last& ymwdl, const years& dy) noexcept;
constexpr year_month_weekday_last
  operator+(const years& dy, const year_month_weekday_last& ymwdl) noexcept;
constexpr year_month_weekday_last
  operator-(const year_month_weekday_last& ymwdl, const months& dm) noexcept;
constexpr year_month_weekday_last
  operator-(const year_month_weekday_last& ymwdl, const years& dy) noexcept;

template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const year_month_weekday_last& ymwdl);
constexpr year_month_day
    operator/(const month_day& md, const year& y) noexcept;
constexpr year_month_day
    operator/(const month_day& md, int y) noexcept;
constexpr year_month_day_last
    operator/(const year_month& ym, last_spec) noexcept;
constexpr year_month_day_last
    operator/(const year& y, const month_day_last& mdl) noexcept;
constexpr year_month_day_last
    operator/(int y, const month_day_last& mdl) noexcept;
constexpr year_month_day_last
    operator/(const month_day_last& mdl, const year& y) noexcept;
constexpr year_month_day_last
    operator/(const month_day_last& mdl, int y) noexcept;
constexpr year_month_day_last
    operator/(const year_month& ym, const weekday_indexed& wdi) noexcept;
constexpr year_month_day_last
    operator/(const year& y, const month_weekday& mwd) noexcept;
constexpr year_month_day_last
    operator/(int y, const month_weekday& mwd) noexcept;
constexpr year_month_day_last
    operator/(const month_weekday& mwd, const year& y) noexcept;
constexpr year_month_day_last
    operator/(const month_weekday& mwd, int y) noexcept;
constexpr year_month_weekday
    operator/(const year_month& ym, const weekday_indexed& wdi) noexcept;
constexpr year_month_weekday
    operator/(const year& y, const month_weekday& mwd) noexcept;
constexpr year_month_weekday
    operator/(int y, const month_weekday& mwd) noexcept;
constexpr year_month_weekday
    operator/(const month_weekday& mwd, const year& y) noexcept;
constexpr year_month_weekday
    operator/(const month_weekday& mwd, int y) noexcept;
constexpr year_month_weekday_last
    operator/(const year_month& ym, const weekday_last& wdl) noexcept;
constexpr year_month_weekday_last
    operator/(const year& y, const month_weekday_last& mwdl) noexcept;
constexpr year_month_weekday_last
    operator/(int y, const month_weekday_last& mwdl) noexcept;
constexpr year_month_weekday_last
    operator/(const month_weekday_last& mwdl, const year& y) noexcept;
constexpr year_month_weekday_last
    operator/(const month_weekday_last& mwdl, int y) noexcept;

// 27.9, class template hh_mm_ss
template<class Duration> class hh_mm_ss;

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const hh_mm_ss<Duration>& hms);

// 27.10, 12/24 hour functions
constexpr bool is_am(const hours& h) noexcept;
constexpr bool is_pm(const hours& h) noexcept;
constexpr hours make12(const hours& h, bool is_pm) noexcept;
constexpr hours make24(const hours& h, bool is_pm) noexcept;

// 27.11.2, time zone database
struct tzdb;
    class tzdb_list;

    // 27.11.2.3, time zone database access
    const tzdb& get_tzdb();
    tzdb_list& get_tzdb_list();
    const time_zone* locate_zone(string_view tz_name);
    const time_zone* current_zone();

    // 27.11.2.4, remote time zone database support
    const tzdb& reload_tzdb();
    string remote_version();

    // 27.11.3, exception classes
    class nonexistent_local_time;
class ambiguous_local_time;

// 27.11.4, information classes
struct sys_info;
  template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const sys_info& si);

struct local_info;
  template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const local_info& li);

// 27.11.5, class time_zone
enum class choose {earliest, latest};
class time_zone;

  bool operator==(const time_zone& x, const time_zone& y) noexcept;
  strong_ordering operator<=>(const time_zone& x, const time_zone& y) noexcept;

// 27.11.6, class template zoned_traits
  template<class T> struct zoned_traits;

// 27.11.7, class template zoned_time
  template<class Duration, class TimeZonePtr = const time_zone*> class zoned_time;
  using zoned_seconds = zoned_time<seconds>;
  template<class Duration1, class Duration2, class TimeZonePtr>
  bool operator==(const zoned_time<Duration1, TimeZonePtr>& x,
                  const zoned_time<Duration2, TimeZonePtr>& y);
  template<class charT, class traits, class Duration, class TimeZonePtr>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os,
              const zoned_time<Duration, TimeZonePtr>& t);

// 27.11.8, leap second support
class leap_second;

  bool operator==(const leap_second& x, const leap_second& y);
  strong_ordering operator<=>(const leap_second& x, const leap_second& y);
  template<class Duration>
  bool operator==(const leap_second& x, const sys_time<Duration>& y);
  template<class Duration>
  bool operator< (const leap_second& x, const sys_time<Duration>& y);
  template<class Duration>
  bool operator< (const sys_time<Duration>& x, const leap_second& y);
  template<class Duration>
  bool operator> (const leap_second& x, const sys_time<Duration>& y);
  template<class Duration>
  bool operator> (const sys_time<Duration>& x, const leap_second& y);
  template<class Duration>
  bool operator<= (const leap_second& x, const sys_time<Duration>& y);
  template<class Duration>
  bool operator<= (const sys_time<Duration>& x, const leap_second& y);
  template<class Duration>
  bool operator>= (const leap_second& x, const sys_time<Duration>& y);
  template<class Duration>
  bool operator>= (const sys_time<Duration>& x, const leap_second& y);
  template<class Duration>
  requires three_way_comparable_with<sys_seconds, sys_time<Duration>>
  constexpr auto operator<=>(const leap_second& x, const sys_time<Duration>& y);
// 27.11.9, class time_zone_link
class time_zone_link;

bool operator==(const time_zone_link& x, const time_zone_link& y);
strong_ordering operator<=>(const time_zone_link& x, const time_zone_link& y);

// 27.12, formatting

template<class Duration> struct local_time_format_t;  // exposition only
template<class Duration>
local_time_format_t<Duration>
local_time_format(local_time<Duration> time, const string* abbrev = nullptr,
const seconds* offset_sec = nullptr);

}  // namespace

}  // namespace chrono

namespace chrono {

} // namespace chrono

// 27.13, parsing

unspecified
parse(const basic_string<charT, traits, Alloc>& format, Parsable& tp);

unspecified
parse(const basic_string<charT, traits, Alloc>& format, Parsable& tp,
basic_string<charT, traits, Alloc>& abbrev);
template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& format, Parsable& tp,
minutes& offset);

template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& format, Parsable& tp,
basic_string<charT, traits, Alloc>& abbrev, minutes& offset);

// calendrical constants
inline constexpr last_spec last{};

inline constexpr weekday Sunday{0};
inline constexpr weekday Monday{1};
inline constexpr weekday Tuesday{2};
inline constexpr weekday Wednesday{3};
inline constexpr weekday Thursday{4};
inline constexpr weekday Friday{5};
inline constexpr weekday Saturday{6};

inline constexpr month January{1};
inline constexpr month February{2};
inline constexpr month March{3};
inline constexpr month April{4};
inline constexpr month May{5};
inline constexpr month June{6};
inline constexpr month July{7};
inline constexpr month August{8};
inline constexpr month September{9};
inline constexpr month October{10};
inline constexpr month November{11};
inline constexpr month December{12};

} inline namespace literals {

} inline namespace chrono_literals {

    // 27.5.9, suffixes for duration literals
    constexpr chrono::hours operator""h(unsigned long long);
    constexpr chrono::duration<unspecified, ratio<3600, 1>> operator""h(long double);
    constexpr chrono::minutes operator""min(unsigned long long);
    constexpr chrono::duration<unspecified, ratio<60, 1>> operator""min(long double);
    constexpr chrono::seconds operator""s(unsigned long long);
    constexpr chrono::duration<unspecified> operator""s(long double);
    constexpr chrono::milliseconds operator""ms(unsigned long long);
    constexpr chrono::duration<unspecified, milli> operator""ms(long double);
    constexpr chrono::microseconds operator""us(unsigned long long);
    constexpr chrono::duration<unspecified, micro> operator""us(long double);
    constexpr chrono::nanoseconds operator""ns(unsigned long long);
    constexpr chrono::duration<unspecified, nano> operator""ns(long double);

    // 27.8.3.3, non-member functions
    constexpr chrono::day operator""d(unsigned long long d) noexcept;

    // 27.8.5.3, non-member functions
    constexpr chrono::year operator""y(unsigned long long y) noexcept;
}
namespace chrono {
    using namespace literals::chrono_literals;
}

27.3 Cpp17Clock requirements

A clock is a bundle consisting of a duration, a time_point, and a function now() to get the current time_point. The origin of the clock’s time_point is referred to as the clock’s epoch. A clock shall meet the requirements in Table 97.

In Table 97 C1 and C2 denote clock types. t1 and t2 are values returned by C1::now() where the call returning t1 happens before (6.9.2) the call returning t2 and both of these calls occur before C1::time_point::max().

[Note 1: This means C1 did not wrap around between t1 and t2. — end note]

Table 97: Cpp17Clock requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1::rep</td>
<td>An arithmetic type or a class emulating an arithmetic type</td>
<td>The representation type of C1::duration.</td>
</tr>
<tr>
<td>C1::period</td>
<td>a specialization of ratio</td>
<td>The tick period of the clock in seconds.</td>
</tr>
<tr>
<td>C1::duration</td>
<td>chrono::duration&lt;C1::rep, C1::period&gt;</td>
<td>The duration type of the clock.</td>
</tr>
<tr>
<td>C1::time_point</td>
<td>chrono::time_point&lt;C1&gt; or chrono::time_point&lt;C2, C1::duration&gt;</td>
<td>The time_point type of the clock. C1 and C2 shall refer to the same epoch.</td>
</tr>
<tr>
<td>C1::is_steady</td>
<td>const bool</td>
<td>true if t1 &lt;= t2 is always true and the time between clock ticks is constant, otherwise false.</td>
</tr>
<tr>
<td>C1::now()</td>
<td>C1::time_point</td>
<td>Returns a time_point object representing the current point in time.</td>
</tr>
</tbody>
</table>

3 [Note 2: The relative difference in durations between those reported by a given clock and the SI definition is a measure of the quality of implementation. — end note]

A type TC meets the Cpp17TrivialClock requirements if:

— TC meets the Cpp17Clock requirements,

— the types TC::rep, TC::duration, and TC::time_point meet the Cpp17EqualityComparable (Table 25) and Cpp17LessThanComparable (Table 26) requirements and the requirements of numeric types (26.2).

[Note 3: This means, in particular, that operations on these types will not throw exceptions. — end note]

— values of the types TC::rep, TC::duration, and TC::time_point are swappable (16.4.4.3),

— the function TC::now() does not throw exceptions, and

— the type TC::time_point::clock meets the Cpp17TrivialClock requirements, recursively.

27.4 Time-related traits

27.4.1 treat_asFloatingPoint

template<class Rep> struct treat_asFloatingPoint : isFloatingPoint<Rep> { };

The duration template uses the treat_asFloatingPoint trait to help determine if a duration object can be converted to another duration with a different tick period. If treat_asFloatingPoint_v<Rep> is true, then implicit conversions are allowed among durations. Otherwise, the implicit convertibility depends on the tick periods of the durations.

[Note 1: The intention of this trait is to indicate whether a given class behaves like a floating-point type, and thus allows division of one value by another with acceptable loss of precision. If treat_asFloatingPoint_v<Rep> is false, Rep will be treated as if it behaved like an integral type for the purpose of these conversions. — end note]
27.4.2 duration_values

The `duration_values` template uses the `duration_values` trait to construct special values of the duration's representation (`Rep`). This is done because the representation can be a class type with behavior that requires some other implementation to return these special values. In that case, the author of that class type should specialize `duration_values` to return the indicated values.

```
static constexpr Rep zero() noexcept;

Returns: Rep(0).
[Note 1: Rep(0) is specified instead of Rep() because Rep() can have some other meaning, such as an uninitialized value. —end note]

Remarks: The value returned shall be the additive identity.
```

```
static constexpr Rep min() noexcept;

Returns: numeric_limits<Rep>::lowest().

Remarks: The value returned shall compare less than or equal to zero().
```

```
static constexpr Rep max() noexcept;

Returns: numeric_limits<Rep>::max().

Remarks: The value returned shall compare greater than zero().
```

27.4.3 Specializations of common_type

```
template<class Rep1, class Period1, class Rep2, class Period2>
struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>> {
    using type = chrono::duration<common_type_t<Rep1, Rep2>, see below>;
};
```

The period of the duration indicated by this specialization of `common_type` is the greatest common divisor of `Period1` and `Period2`.

[Note 1: This can be computed by forming a ratio of the greatest common divisor of `Period1::num` and `Period2::num` and the least common multiple of `Period1::den` and `Period2::den`. —end note]

```
template<class Clock, class Duration1, class Duration2>
struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>> {
    using type = chrono::time_point<Clock, common_type_t<Duration1, Duration2>>;
};
```

The common type of two `time_point` types is a `time_point` with the same clock as the two types and the common type of their two durations.

27.4.4 Class template is_clock

```
template<class T> struct is_clock;
```

`is_clock` is a `Cpp17UnaryTypeTrait` (20.15.2) with a base characteristic of `true_type` if `T` meets the `Cpp17Clock` requirements (27.3), otherwise `false_type`. For the purposes of the specification of this trait, the extent to which an implementation determines that a type cannot meet the `Cpp17Clock` requirements is unspecified, except that as a minimum a type `T` shall not qualify as a `Cpp17Clock` unless it meets all of the following conditions:
— the qualified-ids T::rep, T::period, T::duration, and T::time_point are valid and each denotes a type (13.10.3),

(1.2) — the expression T::is_steady is well-formed when treated as an unevaluated operand,

(1.3) — the expression T::now() is well-formed when treated as an unevaluated operand.

2 The behavior of a program that adds specializations for is_clock is undefined.

27.5 Class template duration

27.5.1 General

A duration type measures time between two points in time (time_points). A duration has a representation which holds a count of ticks and a tick period. The tick period is the amount of time which occurs from one tick to the next, in units of seconds. It is expressed as a rational constant using the template ratio.

```cpp
namespace std::chrono {
    template<class Rep, class Period = ratio<1>>
    class duration {
    public:
        using rep = Rep;
        using period = typename Period::type;

    private:
        rep rep_;  // exposition only

    public:
        // 27.5.2, construct/copy/destroy
        constexpr duration() = default;
        template<class Rep2>
        constexpr explicit duration(const Rep2& r);
        template<class Rep2, class Period2>
        constexpr duration(const duration<Rep2, Period2>& d);
        ~duration() = default;
        duration(duration&) = default;
        duration& operator=(duration&);

        // 27.5.3, observer
        constexpr rep count() const;

        // 27.5.4, arithmetic
        constexpr common_type_t<duration> operator+(const duration&) const;
        constexpr common_type_t<duration> operator-(const duration&) const;
        constexpr duration& operator++();
        constexpr duration& operator++(int);
        constexpr duration& operator--();
        constexpr duration& operator--(int);

        constexpr duration& operator+=(const duration& d);
        constexpr duration& operator-=(const duration& d);

        constexpr duration& operator*=(const rep& rhs);
        constexpr duration& operator/=(const rep& rhs);
        constexpr duration& operator%=(const rep& rhs);
        constexpr duration& operator%=(const duration& rhs);

        // 27.5.5, special values
        static constexpr duration zero() noexcept;
        static constexpr duration min() noexcept;
        static constexpr duration max() noexcept;
    };
}
```

2 Rep shall be an arithmetic type or a class emulating an arithmetic type. If duration is instantiated with a duration type as the argument for the template parameter Rep, the program is ill-formed.
If Period is not a specialization of ratio, the program is ill-formed. If Period::num is not positive, the program is ill-formed.

Members of duration do not throw exceptions other than those thrown by the indicated operations on their representations.

The defaulted copy constructor of duration shall be a constexpr function if and only if the required initialization of the member rep_ for copy and move, respectively, would satisfy the requirements for a constexpr function.

[Example 1:

duration<long, ratio<60>> d0; // holds a count of minutes using a long
duration<long long, milli> d1; // holds a count of milliseconds using a long long
duration<double, ratio<1, 30>> d2; // holds a count with a tick period of \frac{1}{30} of a second
   // (30 Hz) using a double

— end example]

27.5.2 Constructors

template<class Rep2>
constexpr explicit duration(const Rep2& r);

1 Constraints: is_convertible_v<const Rep2&, rep> is true and
(1.1) — treat_as_floating_point_v<rep> is true or
(1.2) — treat_as_floating_point_v<Rep2> is false.

[Example 1:

duration<int, milli> d(3); // OK
duration<int, milli> d(3.5); // error

— end example]

2 Postconditions: count() == static_cast<rep>(r).

template<class Rep2, class Period2>
constexpr duration(const duration<Rep2, Period2>& d);

3 Constraints: No overflow is induced in the conversion and treat_as_floating_point_v<rep> is true or both ratio_divide<Period2, period>::den is 1 and treat_as_floating_point_v<Rep2> is false.

[Note 1: This requirement prevents implicit truncation error when converting between integral-based duration types. Such a construction could easily lead to confusion about the value of the duration. — end note]

[Example 2:

duration<int, milli> ms(3);
duration<int, micro> us = ms; // OK
duration<int, milli> ms2 = us; // error

— end example]

4 Effects: Initializes rep_ with duration_cast<duration>(d).count().

27.5.3 Observer

constexpr rep count() const;

Returns: rep_.

27.5.4 Arithmetic

constexpr common_type_t<duration> operator++() const;

Returns: common_type_t<duration>(*this).

constexpr common_type_t<duration> operator--() const;

Returns: common_type_t<duration>(-rep_).
constexpr duration& operator++();
  Effects: Equivalent to: ++rep_.
  Returns: *this.

constexpr duration operator++(int);
  Effects: Equivalent to: return duration(rep_++);

constexpr duration& operator--();
  Effects: Equivalent to: --rep_.
  Returns: *this.

constexpr duration operator--(int);
  Effects: Equivalent to: return duration(rep_--);

constexpr duration& operator+=(const duration& d);
  Effects: Equivalent to: rep_ += d.count().
  Returns: *this.

constexpr duration& operator-=(const duration& d);
  Effects: Equivalent to: rep_ -= d.count().
  Returns: *this.

constexpr duration& operator*=(const rep& rhs);
  Effects: Equivalent to: rep_ *= rhs.
  Returns: *this.

constexpr duration& operator/=(const rep& rhs);
  Effects: Equivalent to: rep_ /= rhs.
  Returns: *this.

constexpr duration& operator%=(const rep& rhs);
  Effects: Equivalent to: rep_ %= rhs.
  Returns: *this.

constexpr duration& operator%=(const duration& rhs);
  Effects: Equivalent to: rep_ %= rhs.count().
  Returns: *this.

27.5.5 Special values

static constexpr duration zero() noexcept;
  Returns: duration(duration_values<rep>::zero()).

static constexpr duration min() noexcept;
  Returns: duration(duration_values<rep>::min()).

static constexpr duration max() noexcept;
  Returns: duration(duration_values<rep>::max()).

27.5.6 Non-member arithmetic

In the function descriptions that follow, unless stated otherwise, let CD represent the return type of the function.
27.5.7 Comparisons

In the function descriptions that follow, CT represents \( \text{common\_type\_t} < A, B > \), where A and B are the types of the two arguments to the function.

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: \( \text{CD}(\text{CD}(\text{lhs}).\text{count()} + \text{CD}(\text{rhs}).\text{count()}) \).
```

```cpp
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: \( \text{CD}(\text{CD}(\text{lhs}).\text{count()} - \text{CD}(\text{rhs}).\text{count()}) \).
```

```cpp
template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period> operator*(const duration<Rep1, Period>& d, const Rep2& s);

Constraints: is_convertible_v<const Rep2&, common_type_t<Rep1, Rep2>> is true.

Returns: \( \text{CD}(\text{CD}(\text{d}).\text{count()} * \text{s}) \).
```

```cpp
template<class Rep1, class Rep2, class Period>
constexpr duration<common_type_t<Rep1, Rep2>, Period> operator*(const Rep1& s, const duration<Rep2, Period>& d);

Constraints: is_convertible_v<const Rep2&, common_type_t<Rep1, Rep2>> is true and Rep2 is not a specialization of duration.

Returns: \( \text{d} * \text{s} \).
```

```cpp
template<class Rep1, class Rep2, class Period>
constexpr duration<common_type_t<Rep1, Rep2>, Period> operator%(const duration<Rep1, Period>& d, const Rep2& s);

Constraints: is_convertible_v<const Rep2&, common_type_t<Rep1, Rep2>> is true and Rep2 is not a specialization of duration.

Returns: \( \text{CD}(\text{CD}(\text{d}).\text{count()} / \text{s}) \).
```

```cpp
template<class Rep1, class Rep2, class Period1, class Rep2, class Period2>
constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>> operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: \( \text{CD}(\text{CD}(\text{lhs}).\text{count()} % \text{CD}(\text{rhs}).\text{count()}) \).
```

27.5.7 Comparisons
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: CT(lhs).count() < CT(rhs).count().

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: rhs < lhs.

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: !(rhs < lhs).

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: !(lhs < rhs).

template<class Rep1, class Period1, class Rep2, class Period2>
requires three_way_comparable<typename CT::rep>
constexpr auto operator<=>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: CT(lhs).count() <=> CT(rhs).count().

27.5.8 Conversions [time.duration.cast]

template<class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);

Constraints: ToDuration is a specialization of duration.

Returns: Let CF be ratio_divide<Period, typename ToDuration::period>, and CR be common_-type<typename ToDuration::rep, Rep, intmax_t>::type.

(2.1) — If CF::num == 1 and CF::den == 1, returns
ToDuration(static_cast<typename ToDuration::rep>(d.count()))
(2.2) — otherwise, if CF::num != 1 and CF::den == 1, returns
ToDuration(static_cast<typename ToDuration::rep>(
static_cast<CR>(d.count()) * static_cast<CR>(CF::num)))
(2.3) — otherwise, if CF::num == 1 and CF::den != 1, returns
ToDuration(static_cast<typename ToDuration::rep>(
static_cast<CR>(d.count()) / static_cast<CR>(CF::den)))
(2.4) — otherwise, returns
ToDuration(static_cast<typename ToDuration::rep>(
static_cast<CR>(d.count()) * static_cast<CR>(CF::num) / static_cast<CR>(CF::den)))

[Note 1: This function does not use any implicit conversions; all conversions are done with static_cast. It avoids multiplications and divisions when it is known at compile time that one or more arguments is 1. Intermediate computations are carried out in the widest representation and only converted to the destination representation at the final step. — end note]

template<class ToDuration, class Rep, class Period>
constexpr ToDuration floor(const duration<Rep, Period>& d);

Constraints: ToDuration is a specialization of duration.

Returns: The greatest result t representable in ToDuration for which t <= d.

template<class ToDuration, class Rep, class Period>
constexpr ToDuration ceil(const duration<Rep, Period>& d);

Constraints: ToDuration is a specialization of duration.
Returns: The least result t representable in ToDuration for which t >= d.

template<class ToDuration, class Rep, class Period>
constexpr ToDuration round(const duration<Rep, Period>& d);

Constraints: ToDuration is a specialization of duration and treat_as_floating_point_v<typename ToDuration::rep> is false.

Returns: The value of ToDuration that is closest to d. If there are two closest values, then return the value t for which t % 2 == 0.

27.5.9 Suffixes for duration literals

This subclause describes literal suffixes for constructing duration literals. The suffixes h, min, s, ms, us, ns denote duration values of the corresponding types hours, minutes, seconds, milliseconds, microseconds, and nanoseconds respectively if they are applied to integer-literals.

If any of these suffixes are applied to a floating-point-literal the result is a chrono::duration literal with an unspecified floating-point representation.

If any of these suffixes are applied to an integer-literal and the resulting chrono::duration value cannot be represented in the result type because of overflow, the program is ill-formed.

Example 1: The following code shows some duration literals.

```cpp
using namespace std::chrono_literals;
auto constexpr aday=24h;
auto constexpr lesson=45min;
auto constexpr halfanhour=0.5h;
```

Returns: A duration literal representing hours.

Returns: A duration literal representing minutes.

Returns: A duration literal representing sec seconds.

Note 1: The same suffix s is used for basic_string but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — end note

Returns: A duration literal representing msec milliseconds.

Returns: A duration literal representing usec microseconds.

Returns: A duration literal representing nsec nanoseconds.

27.5.10 Algorithms

template<class Rep, class Period>
constexpr duration<Rep, Period> abs(duration<Rep, Period> d);

Constraints: numeric_limits<Rep>::is_signed is true.

Returns: If d >= d.zero(), return d, otherwise return -d.
27.5.11  I/O

template<class charT, class traits, class Rep, class Period>
    basic_ostream<charT, traits>&
        operator<(basic_ostream<charT, traits>& os, const duration<Rep, Period>& d);

1 Effects: Inserts the duration d onto the stream os as if it were implemented as follows:

   basic_ostringstream<charT, traits> s;
   s.flags(os.flags());
   s.imbue(os.getloc());
   s.precision(os.precision());
   s << d.count() << units-suffix;
   return os << s.str();

   where units-suffix depends on the type Period::type as follows:

   — If Period::type is attos, units-suffix is "as".
   — Otherwise, if Period::type is femtos, units-suffix is "fs".
   — Otherwise, if Period::type is picos, units-suffix is "ps".
   — Otherwise, if Period::type is nanos, units-suffix is "ns".
   — Otherwise, if Period::type is micros, it is implementation-defined whether units-suffix is "µs" ("\u00b5\u0073") or "us".
   — Otherwise, if Period::type is millis, units-suffix is "ms".
   — Otherwise, if Period::type is centis, units-suffix is "cs".
   — Otherwise, if Period::type is deca, units-suffix is "ds".
   — Otherwise, if Period::type is hectos, units-suffix is "hs".
   — Otherwise, if Period::type is kilos, units-suffix is "ks".
   — Otherwise, if Period::type is megas, units-suffix is "Ms".
   — Otherwise, if Period::type is gigas, units-suffix is "Gs".
   — Otherwise, if Period::type is teras, units-suffix is "Ts".
   — Otherwise, if Period::type is petas, units-suffix is "Ps".
   — Otherwise, if Period::type is exas, units-suffix is "Es".
   — Otherwise, if Period::type is ratio<60>, units-suffix is "min".
   — Otherwise, if Period::type is ratio<3600>, units-suffix is "h".
   — Otherwise, if Period::type is ratio<86400>, units-suffix is "d".
   — Otherwise, if Period::type::den == 1, units-suffix is "[num/1]s".
   — Otherwise, units-suffix is "[num/den]s".

In the list above, the use of num and den refer to the static data members of Period::type, which are converted to arrays of charT using a decimal conversion with no leading zeroes.

2 Returns: os.

template<class charT, class traits, class Rep, class Period, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
        from_stream(basic_istream<charT, traits>& is, const charT* fmt, duration<Rep, Period>& d, basic_string<charT, traits, Alloc>* abbrev = nullptr, minutes* offset = nullptr);

3 Effects: Attempts to parse the input stream is into the duration d using the format flags given in the NTCTS fmt as specified in 27.13. If the parse parses everything specified by the parsing format flags without error, and yet none of the flags impacts a duration, d will be assigned a zero value. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a
modified variant) is used and successfully parsed, that value will be assigned to \*offset if offset is non-null.

*Returns*: 

27.6 Class template time_point

27.6.1 General

namespace std::chrono {
  template<class Clock, class Duration = typename Clock::duration>
  class time_point {
public:
    using clock = Clock;
    using duration = Duration;
    using rep = typename duration::rep;
    using period = typename duration::period;

private:
  duration d_; // exposition only

public:
  // 27.6.2, construct
  constexpr time_point(); // has value epoch
  constexpr explicit time_point(const duration& d); // same as time_point() + d
  template<class Duration2>
  constexpr time_point(const time_point<clock, Duration2>& t);

  // 27.6.3, observer
  constexpr duration time_since_epoch() const;

  // 27.6.4, arithmetic
  constexpr time_point& operator++();
  constexpr time_point operator++(int);
  constexpr time_point& operator--();
  constexpr time_point operator--(int);
  constexpr time_point& operator+=(const duration& d);
  constexpr time_point& operator-=(const duration& d);

  // 27.6.5, special values
  static constexpr time_point min() noexcept;
  static constexpr time_point max() noexcept;
};
}

Clock shall either meet the Cpp17Clock requirements (27.3) or be the type local_t.

If Duration is not an instance of duration, the program is ill-formed.

27.6.2 Constructors

constexpr time_point();

*Effects*: Initializes d_ with duration::zero(). Such a time_point object represents the epoch.

constexpr explicit time_point(const duration& d);

*Effects*: Initializes d_ with d. Such a time_point object represents the epoch + d.

template<class Duration2>
constexpr time_point(const time_point<clock, Duration2>& t);

*Constraints*: is_convertible_v<Duration2, duration> is true.

*Effects*: Initializes d_ with t.time_since_epoch().
27.6.3 Observer

```cpp
constexpr duration time_since_epoch() const;
```

Returns: \( d_\).

27.6.4 Arithmetic

```cpp
constexpr time_point& operator++();

Effects: Equivalent to: ++\( d_\).

Returns: \*this.
```

```cpp
constexpr time_point& operator++(int);

Effects: Equivalent to: return time_point(d_++);

Returns: \*this.
```

```cpp
constexpr time_point operator--();

Effects: Equivalent to: \( \text{return time_point}(d_-) \).

Returns: \*this.
```

```cpp
constexpr time_point operator--(int);

Effects: Equivalent to: \( \text{return time_point}(d_-) \).

Returns: \*this.
```

```cpp
constexpr time_point& operator+=(const duration& d);

Effects: Equivalent to: \( d_+ d \).

Returns: \*this.
```

```cpp
constexpr time_point& operator-=(const duration& d);

Effects: Equivalent to: \( d_ - d \).

Returns: \*this.
```

27.6.5 Special values

```cpp
static constexpr time_point min() noexcept;

Returns: time_point(duration::min()).
```

```cpp
static constexpr time_point max() noexcept;

Returns: time_point(duration::max()).
```

27.6.6 Non-member arithmetic

```cpp
template<class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>> operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: \( \text{CT}(\text{lhs.time_since_epoch()} + \text{rhs}) \), where \( \text{CT} \) is the type of the return value.
```

```cpp
template<class Rep1, class Period1, class Clock, class Duration2>
constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>> operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);

Returns: \( \text{rhs} + \text{lhs} \).
```

```cpp
template<class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, duration<Rep2, Period2>> operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

Returns: \( \text{CT}(\text{lhs.time_since_epoch()} - \text{rhs}) \), where \( \text{CT} \) is the type of the return value.
```

```cpp
template<class Clock, class Duration1, class Duration2>
constexpr common_type_t<Duration1, Duration2> operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

Returns: \( \text{lhs.time_since_epoch()} - \text{rhs.time_since_epoch()} \).
```
27.6.7 Comparisons

```cpp
template<class Clock, class Duration1, class Duration2>
constexpr bool operator==(const time_point<Clock, Duration1>& lhs, 
const time_point<Clock, Duration2>& rhs);
```

1. Returns: \( \text{lhs.time_since_epoch()} == \text{rhs.time_since_epoch}() \).

```cpp
template<class Clock, class Duration1, class Duration2>
constexpr bool operator<(const time_point<Clock, Duration1>& lhs, 
const time_point<Clock, Duration2>& rhs);
```

2. Returns: \( \text{lhs.time_since_epoch()} < \text{rhs.time_since_epoch}() \).

```cpp
template<class Clock, class Duration1, class Duration2>
constexpr bool operator>(const time_point<Clock, Duration1>& lhs, 
const time_point<Clock, Duration2>& rhs);
```

3. Returns: \( \text{rhs < lhs} \).

```cpp
template<class Clock, class Duration1, class Duration2>
constexpr bool operator<=(const time_point<Clock, Duration1>& lhs, 
const time_point<Clock, Duration2>& rhs);
```

4. Returns: \( \neg(\text{rhs < lhs}) \).

```cpp
template<class Clock, class Duration1, class Duration2>
constexpr bool operator>=(const time_point<Clock, Duration1>& lhs, 
const time_point<Clock, Duration2>& rhs);
```

5. Returns: \( \neg(\text{lhs < rhs}) \).

```cpp
template<class Clock, class Duration1, class Duration2>
constexpr auto operator<=>(const time_point<Clock, Duration1>& lhs, 
const time_point<Clock, Duration2>& rhs);
```

6. Returns: \( \text{lhs.time_since_epoch()} <= \text{rhs.time_since_epoch}() \).

27.6.8 Conversions

```cpp
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);
```

1. Constraints: \( \text{ToDuration} \) is a specialization of \( \text{duration} \).

2. Returns:
   \( \text{time_point<Clock, ToDuration>}(\text{duration_cast<ToDuration>(t.time_since_epoch())}) \)

```cpp
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> floor(const time_point<Clock, Duration>& tp);
```

3. Constraints: \( \text{ToDuration} \) is a specialization of \( \text{duration} \).

4. Returns: \( \text{time_point<Clock, ToDuration>}(\text{floor<ToDuration>(tp.time_since_epoch())}) \)

```cpp
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> ceil(const time_point<Clock, Duration>& tp);
```

5. Constraints: \( \text{ToDuration} \) is a specialization of \( \text{duration} \).

6. Returns: \( \text{time_point<Clock, ToDuration>}(\text{ceil<ToDuration>(tp.time_since_epoch())}) \)

```cpp
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> round(const time_point<Clock, Duration>& tp);
```

7. Constraints: \( \text{ToDuration} \) is a specialization of \( \text{duration} \), and \( \text{treat_as_floating_point_v<\text{typename ToDuration::rep}>} \) is false.

8. Returns: \( \text{time_point<Clock, ToDuration>}(\text{round<ToDuration>(tp.time_since_epoch())}) \).
27.7 Clocks

27.7.1 General

The types defined in 27.7 meet the Cpp17TrivialClock requirements (27.3) unless otherwise specified.

27.7.2 Class system_clock

namespace std::chrono {
    class system_clock {
    public:
        using rep = unspecified;
        using period = ratio<unspecified, unspecified>;
        using duration = chrono::duration<rep, period>;
        using time_point = chrono::time_point<system_clock>;
        static constexpr bool is_steady = unspecified;
        static time_point now() noexcept;

        // mapping to/from C type time_t
        static time_t to_time_t(const time_point& t) noexcept;
        static time_point from_time_t(time_t t) noexcept;
    };
}

1 Objects of type system_clock represent wall clock time from the system-wide realtime clock. Objects of type sys_time<Duration> measure time since 1970-01-01 00:00:00 UTC excluding leap seconds. This measure is commonly referred to as Unix time. This measure facilitates an efficient mapping between sys_time and calendar types (27.8).

[Example 1:
sys_seconds{sys_days{1970y/January/1}}.time_since_epoch() is 0s.
sys_seconds{sys_days{2000y/January/1}}.time_since_epoch() is 946'684'800s, which is 10'957 * 86'400s. —end example]

27.7.2.2 Members

using system_clock::rep = unspecified;

1 Constraints: system_clock::duration::min() < system_clock::duration::zero() is true.
[Note 1: This implies that rep is a signed type. —end note]

static time_t to_time_t(const time_point& t) noexcept;

2 Returns: A time_t object that represents the same point in time as t when both values are restricted to the coarser of the precisions of time_t and time_point. It is implementation-defined whether values are rounded or truncated to the required precision.

static time_point from_time_t(time_t t) noexcept;

3 Returns: A time_point object that represents the same point in time as t when both values are restricted to the coarser of the precisions of time_t and time_point. It is implementation-defined whether values are rounded or truncated to the required precision.

27.7.2.3 Non-member functions

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const sys_time<Duration>& tp);

1 Constraints: treat_asFloatingPoint_v<typename Duration::rep> is false, and Duration{1} < days{1} is true.

2 Effects: Equivalent to:
    auto const dp = floor<days>(tp);
    return os << format(os.getloc(), STATICALLY_WIDEN<charT>("{} {}"),
                        year_month_day(dp), hh_mm_ss(tp-dp));
3  [Example 1:
  
  cout << sys_seconds{0s} << '\n'; // 1970-01-01 00:00:00
  cout << sys_seconds{946'684'800s} << '\n'; // 2000-01-01 00:00:00
  cout << sys_seconds{946'688'523s} << '\n'; // 2000-01-01 01:02:03
  —end example]  

4  template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const sys_days& dp);

5  Effects: os << year_month_day{dp}.

6  Returns: os.

7  template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
  from_stream(basic_istream<charT, traits>& is, const charT* fmt,
              sys_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
              minutes* offset = nullptr);

6  Effects: Attempts to parse the input stream is into the sys_time tp using the format flags given in
the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) is called and tp is not modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null. Additionally, the parsed offset will be subtracted from the successfully parsed timestamp prior to assigning that difference to tp.

7  Returns: is.

27.7.3 Class utc_clock

27.7.3.1 Overview

namespace std::chrono {
  class utc_clock {
  public:
    using rep = a signed arithmetic type;
    using period = ratio<unspecified, unspecified>;
    using duration = chrono::duration<rep, period>;
    using time_point = chrono::time_point<utc_clock>;
    static constexpr bool is_steady = unspecified;
    static time_point now();
  
  template<class Duration>
    static sys_time<common_type_t<Duration, seconds>>
    to_sys(const utc_time<Duration>& t);
  template<class Duration>
    static utc_time<common_type_t<Duration, seconds>>
    from_sys(const sys_time<Duration>& t);
  
  
  };
}

1  In contrast to sys_time, which does not take leap seconds into account, utc_clock and its associated
  time_point, utc_time, count time, including leap seconds, since 1970-01-01 00:00:00 UTC.

[Note 1: The UTC time standard began on 1972-01-01 00:00:10 TAI. To measure time since this epoch instead, one
  can add/subtract the constant sys_days{1972y/1/1} - sys_days{1970y/1/1} (63'072'000s) from the utc_time.
  —end note]

[Example 1:
  clock_cast<utc_clock>(sys_seconds{sys_days{1972y/January/1}}).time_since_epoch() is 0s.
  clock_cast<utc_clock>(sys_seconds{sys_days{2000y/January/1}}).time_since_epoch() is 946'684'822s,
  which is 10'957 * 86'400s + 22s.
  —end example]

2  utc_clock is not a Cpp17TrivialClock unless the implementation can guarantee that utc_clock::now() does not propagate an exception.

§ 27.7.3.1
[Note 2: noexcept(from_sys(system_clock::now())) is false. — end note]

27.7.3.2 Member functions

static time_point now();

1  Returns: from_sys(system_clock::now()), or a more accurate value of utc_time.

template<class Duration>
static sys_time<common_type_t<Duration, seconds>>
to_sys(const utc_time<Duration>& u);

2  Returns: A sys_time t such that from_sys(t) == u if such a mapping exists. Otherwise u represents a time_point during a positive leap second insertion, the conversion counts that leap second as not inserted, and the last representable value of sys_time prior to the insertion of the leap second is returned.

template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
from_sys(const sys_time<Duration>& t);

3  Returns: A utc_time u such that u.time_since_epoch() - t.time_since_epoch() is equal to the sum of leap seconds that were inserted between t and 1970-01-01. If t is exactly the date of leap second insertion, then the conversion counts that leap second as inserted.

[Example 1:
  auto t = sys_days{July/1/2015} - 2ns;
  auto u = utc_clock::from_sys(t);
  assert(u.time_since_epoch() - t.time_since_epoch() == 25s);
  t += 1ns;
  u = utc_clock::from_sys(t);
  assert(u.time_since_epoch() - t.time_since_epoch() == 25s);
  t += 1ns;
  t = utc_clock::from_sys(t);
  assert(u.time_since_epoch() - t.time_since_epoch() == 26s);
  t += 1ns;
  u = utc_clock::from_sys(t);
  assert(u.time_since_epoch() - t.time_since_epoch() == 26s);
—end example]

27.7.3.3 Non-member functions

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const utc_time<Duration>& t);

1  Effects: Equivalent to:

        return os << format(STATICALLY-WIDEN<charT>("{%F %T}"), t);

2  [Example 1:
  auto t = sys_days{July/1/2015} - 500ms;
  auto u = clock_cast<utc_clock>(t);
  for (auto i = 0; i < 8; ++i, u += 250ms)
    cout << u << " UTC\n";

  Produces this output:
  2015-06-30 23:59:59.500 UTC
  2015-06-30 23:59:59.750 UTC
  2015-06-30 23:59:60.000 UTC
  2015-06-30 23:59:60.250 UTC
  2015-06-30 23:59:60.500 UTC
  2015-06-30 23:59:60.750 UTC
  2015-07-01 00:00:00.000 UTC
  2015-07-01 00:00:00.250 UTC

—end example]
template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
utc_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the utc_time tp using the format flags given in
the NTCIS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_-base::failbit) is called and tp is not modified. If %Z is used and successfully parsed, that value will
be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully
parsed, that value will be assigned to *offset if offset is non-null. Additionally, the parsed offset
will be subtracted from the successfully parsed timestamp prior to assigning that difference to tp.

Returns: is.

struct leap_second_info {
  bool  is_leap_second;
  seconds elapsed;
};

The type leap_second_info has data members and special members specified above. It has no base
classes or members other than those specified.

template<class Duration>
leap_second_info get_leap_second_info(const utc_time<Duration>& ut);

Returns: A leap_second_info lsi, where lsi.is_leap_second is true if ut is during a positive leap
second insertion, and otherwise false. lsi.elapsed is the sum of leap seconds between 1970-01-01
and ut. If lsi.is_leap_second is true, the leap second referred to by ut is included in the sum.

27.7.4 Class tai_clock

27.7.4.1 Overview

namespace std::chrono {
  class tai_clock {
    public:
      using rep = a signed arithmetic type;
      using period = ratio<unspecified, unspecified>;
      using duration = chrono::duration<rep, period>;
      using time_point = chrono::time_point<tai_clock>;
      static constexpr bool is_steady = unspecified;
      static time_point now();
  };
}

The clock tai_clock measures seconds since 1958-01-01 00:00:00 and is offset 10s ahead of UTC at this
date. That is, 1958-01-01 00:00:00 TAI is equivalent to 1957-12-31 23:59:50 UTC. Leap seconds are not
inserted into TAI. Therefore every time a leap second is inserted into UTC, UTC shifts another second with
respect to TAI. For example by 2000-01-01 there had been 22 positive and 0 negative leap seconds inserted
so 2000-01-01 00:00:00 UTC is equivalent to 2000-01-01 00:00:32 TAI (22s plus the initial 10s offset).

tai_clock is not a Cpp17TrivialClock unless the implementation can guarantee that tai_clock::now() does not propagate an exception.

[Note 1: noexcept(from_utc(utc_clock::now())) is false. —end note]
27.7.4.2 Member functions

static time_point now();

Returns: from_utc(utc_clock::now()), or a more accurate value of tai_time.

template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
to_utc(const tai_time<Duration>& t) noexcept;

Returns:

tc_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 378691210s

[Note 1:
378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s
—end note]

tai_time<common_type_t<Duration, seconds>>
from_utc(const utc_time<Duration>& t) noexcept;

Returns:

tai_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} + 378691210s

[Note 2:
378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s
—end note]

27.7.4.3 Non-member functions

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const tai_time<Duration>& t);

Effects: Equivalent to:

return os << format(STATICALLY-WIDEN<charT>("{:%F %T}"), t);

[Example 1:
auto st = sys_days{2000y/January/1};
auto tt = clock_cast<tai_clock>(st);
cout << format("{0:%F %T %Z} == {1:%F %T %Z}\n", st, tt);
Produces this output:
2000-01-01 00:00:00 UTC == 2000-01-01 00:00:32 TAI
—end example]

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
tai_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the tai_time tp using the format flags given in
the NTCITS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) is called and tp is not modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null. Additionally, the parsed offset will be subtracted from the successfully parsed timestamp prior to assigning that difference to tp.

Returns: is.

27.7.5 Class gps_clock

27.7.5.1 Overview

namespace std::chrono {
    class gps_clock {
public:
    using rep = a signed arithmetic type;
    using period = ratio<unspecified, unspecified>;
    using duration = chrono::duration<rep, period>;
    using time_point = chrono::time_point<gps_clock>;
    static constexpr bool is_steady = unspecified;

    static time_point now();

    template<class Duration>
    static utc_time<common_type_t<Duration, seconds>>
        to_utc(const gps_time<Duration>& t) noexcept;

    template<class Duration>
    static gps_time<common_type_t<Duration, seconds>>
        from_utc(const utc_time<Duration>& t) noexcept;
};

1 The clock gps_clock measures seconds since the first Sunday of January, 1980 00:00:00 UTC. Leap seconds are not inserted into GPS. Therefore every time a leap second is inserted into UTC, UTC shifts another second with respect to GPS. Aside from the offset from 1958y/January/1 to 1980y/January/Sunday[1], GPS is behind TAI by 19s due to the 10s offset between 1958 and 1970 and the additional 9 leap seconds inserted between 1970 and 1980.

2 gps_clock is not a Cpp17TrivialClock unless the implementation can guarantee that gps_clock::now() does not propagate an exception.

[Note 1: noexcept(from_utc(utc_clock::now())) is false. — end note]

27.7.5.2 Member functions

static time_point now();
    Returns: from_utc(utc_clock::now()), or a more accurate value of gps_time.

    template<class Duration>
    static utc_time<common_type_t<Duration, seconds>>
        to_utc(const gps_time<Duration>& t) noexcept;

2 Returns:
    gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} + 315964809s
    [Note 1:
        315964809s == sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1} + 9s
        — end note]

    template<class Duration>
    static gps_time<common_type_t<Duration, seconds>>
        from_utc(const utc_time<Duration>& t) noexcept;

3 Returns:
    gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 315964809s
    [Note 2:
        315964809s == sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1} + 9s
        — end note]

27.7.5.3 Non-member functions

    template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const gps_time<Duration>& t);

1 Effects: Equivalent to:
    return os << format(STATICALLY-WIDEN<charT>(":%F %T"), t);

[Example 1:
auto st = sys_days{2000y/January/1};
auto gt = clock_cast<gps_clock>(st);
cout << format("{0:%F %T %Z} == {1:%F %T %Z}n");
st, gt);

Produces this output:
2000-01-01 00:00:00 UTC == 2000-01-01 00:00:13 GPS

—end example

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
gps_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the gps_time tp using the format flags given in
the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_-
base::failbit) is called and tp is not modified. If %Z is used and successfully parsed, that value will
be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully
parsed, that value will be assigned to *offset if offset is non-null. Additionally, the parsed offset
will be subtracted from the successfully parsed timestamp prior to assigning that difference to tp.

Returns: is.

27.7.6 Type file_clock

27.7.6.1 Overview

file_clock is an alias for a type meeting the Cpp17TrivialClock requirements (27.3), and using a signed
arithmetic type for file_clock::rep. file_clock is used to create the time_point system used for
file_time_type (29.11). Its epoch is unspecified, and noexcept(file_clock::now()) is true.

[Note 1: The type that file_clock denotes can be in a different namespace than std::chrono, such as std::file-
system. —end note]

27.7.6.2 Member functions

The type denoted by file_clock provides precisely one of the following two sets of static member functions:

template<class Duration>
static sys_time<see below>
to_sys(const file_time<Duration>&);
template<class Duration>
static file_time<see below>
from_sys(const sys_time<Duration>&);
or:

template<class Duration>
static utc_time<see below>
to_utc(const file_time<Duration>&);
template<class Duration>
static file_time<see below>
from_utc(const utc_time<Duration>&);

These member functions shall provide time_point conversions consistent with those specified by utc_clock,
tai_clock, and gps_clock. The Duration of the resultant time_point is computed from the Duration of
the input time_point.

27.7.6.3 Non-member functions

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const file_time<Duration>& t);

Effects: Equivalent to:

return os << format(STATICALLY-WIDEN<charT>("{:%F %T}"), t);
template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
        file_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
        minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the file_time tp using the format flags given
in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) is called and tp is not modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null. Additionally, the parsed offset will be subtracted from the successfully parsed timestamp prior to assigning that difference to tp.

Returns: is.

27.7.7 Class steady_clock

namespace std::chrono {
    class steady_clock {
    public:
        using rep = unspecified;
        using period = ratio<unspecified, unspecified>;
        using duration = chrono::duration<rep, period>;
        using time_point = chrono::time_point<unspecified, duration>;
        static constexpr bool is_steady = true;

        static time_point now() noexcept;
    };
}

Objects of class steady_clock represent clocks for which values of time_point never decrease as physical time advances and for which values of time_point advance at a steady rate relative to real time. That is, the clock may not be adjusted.

27.7.8 Class high_resolution_clock

namespace std::chrono {
    class high_resolution_clock {
    public:
        using rep = unspecified;
        using period = ratio<unspecified, unspecified>;
        using duration = chrono::duration<rep, period>;
        using time_point = chrono::time_point<unspecified, duration>;
        static constexpr bool is_steady = unspecified;
        static time_point now() noexcept;
    };
}

Objects of class high_resolution_clock represent clocks with the shortest tick period. high_resolution_clock may be a synonym for system_clock or steady_clock.

27.7.9 Local time

The family of time points denoted by local_time<Duration> are based on the pseudo clock local_t.
local_t has no member now() and thus does not meet the clock requirements. Nevertheless local_time<Duration> serves the vital role of representing local time with respect to a not-yet-specified time zone. Aside from being able to get the current time, the complete time_point algebra is available for local_time<Duration> (just as for sys_time<Duration>).

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const local_time<Duration>& lt);

Effects:
    os << sys_time<Duration>{lt.time_since_epoch()};

Returns: os.
template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                local_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

4 Effects: Attempts to parse the input stream is into the local_time tp using the format flags given
in the NICTS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) is called and tp is not modified. If %Z is used and successfully parsed, that value will
be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully
parsed, that value will be assigned to *offset if offset is non-null.

5 Returns: is.

27.7.10 time_point conversions

27.7.10.1 Class template clock_time_conversion

    namespace std::chrono {
        template<class DestClock, class SourceClock>
        struct clock_time_conversion {};
    }

1 clock_time_conversion serves as a trait which can be used to specify how to convert a source time_point of
type time_point<SourceClock, Duration> to a destination time_point of type time_point<DestClock, Duration>
via a specialization: clock_time_conversion<DestClock, SourceClock>. A specialization
of clock_time_conversion<DestClock, SourceClock> shall provide a const-qualified operator() that
takes a parameter of type time_point<SourceClock, Duration> and returns a time_point<DestClock,
OtherDuration> representing an equivalent point in time. OtherDuration is a chrono::duration whose
specialization is computed from the input Duration in a manner which can vary for each clock_time_-
conversion specialization. A program may specialize clock_time_conversion if at least one of the template
parameters is a user-defined clock type.

2 Several specializations are provided by the implementation, as described in 27.7.10.2, 27.7.10.3, 27.7.10.4,
and 27.7.10.5.

27.7.10.2 Identity conversions

    template<class Clock>
    struct clock_time_conversion<Clock, Clock> {
        template<class Duration>
        time_point<Clock, Duration>
            operator()(const time_point<Clock, Duration>& t) const;
    };

    template<class Duration>
    time_point<Clock, Duration>
        operator()(const time_point<Clock, Duration>& t) const;

1 Returns: t.

    template<>
    struct clock_time_conversion<system_clock, system_clock> {
        template<class Duration>
        sys_time<Duration>
            operator()(const sys_time<Duration>& t) const;
    };

    template<class Duration>
    sys_time<Duration>
        operator()(const sys_time<Duration>& t) const;

2 Returns: t.

    template<>
    struct clock_time_conversion<utc_clock, utc_clock> {
        template<class Duration>
        utc_time<Duration>
            operator()(const utc_time<Duration>& t) const;
    };

    template<>
    struct clock_time_conversion<system_clock, utc_clock> {
        template<class Duration>
        sys_time<Duration>
            operator()(const sys_time<Duration>& t) const;
    };

    template<>
    struct clock_time_conversion<utc_clock, system_clock> {
        template<class Duration>
        utc_time<Duration>
            operator()(const utc_time<Duration>& t) const;
    };
template<class Duration>
utc_time<Duration>
operator()(const utc_time<Duration>& t) const;

Returns: t.

27.7.10.3 Conversions between system_clock and utc_clock

[time.clock.cast.sys.utc]

template<>
struct clock_time_conversion<utc_clock, system_clock> {
  template<class Duration>
  utc_time<common_type_t<Duration, seconds>>
  operator()(const sys_time<Duration>& t) const;
};

template<class Duration>
utc_time<common_type_t<Duration, seconds>>
operator()(const sys_time<Duration>& t) const;

Returns: utc_clock::from_sys(t).

template<>
struct clock_time_conversion<system_clock, utc_clock> {
  template<class Duration>
  sys_time<common_type_t<Duration, seconds>>
  operator()(const utc_time<Duration>& t) const;
};

template<class Duration>
sys_time<common_type_t<Duration, seconds>>
operator()(const utc_time<Duration>& t) const;

Returns: utc_clock::to_sys(t).

27.7.10.4 Conversions between system_clock and other clocks

[time.clock.cast.sys]

template<class SourceClock>
struct clock_time_conversion<system_clock, SourceClock> {
  template<class Duration>
  auto operator()(const time_point<SourceClock, Duration>& t) const
  -> decltype(SourceClock::to_sys(t));
};

template<class Duration>
auto operator()(const time_point<SourceClock, Duration>& t) const
-> decltype(SourceClock::to_sys(t));

Constraints: SourceClock::to_sys(t) is well-formed.

Mandates: SourceClock::to_sys(t) returns a sys_time<Duration>, where Duration is a valid
chrono::duration specialization.

Returns: SourceClock::to_sys(t).

template<class DestClock>
struct clock_time_conversion<DestClock, system_clock> {
  template<class Duration>
  auto operator()(const sys_time<Duration>& t) const
  -> decltype(DestClock::from_sys(t));
};

template<class Duration>
auto operator()(const sys_time<Duration>& t) const
-> decltype(DestClock::from_sys(t));

Constraints: DestClock::from_sys(t) is well-formed.

Mandates: DestClock::from_sys(t) returns a time_point<DestClock, Duration>, where Duration
is a valid chrono::duration specialization.

Returns: DestClock::from_sys(t).
27.7.10.5 Conversions between utc_clock and other clocks

```cpp
template<class SourceClock>
struct clock_time_conversion<utc_clock, SourceClock> {
    template<class Duration>
    auto operator()(const time_point<SourceClock, Duration>& t) const
    -> decltype(SourceClock::to_utc(t));
};
```

```cpp
template<class Duration>
auto operator()(const time_point<SourceClock, Duration>& t) const
-> decltype(SourceClock::to_utc(t));
```

1. Constraints: SourceClock::to_utc(t) is well-formed.
2. Mandates: SourceClock::to_utc(t) returns a utc_time<Duration>, where Duration is a valid
chrono::duration specialization.
3. Returns: SourceClock::to_utc(t).

```cpp
template<class DestClock>
struct clock_time_conversion<DestClock, utc_clock> {
    template<class Duration>
    auto operator()(const utc_time<Duration>& t) const
    -> decltype(DestClock::from_utc(t));
};
```

```cpp
template<class Duration>
auto operator()(const utc_time<Duration>& t) const
-> decltype(DestClock::from_utc(t));
```

4. Constraints: DestClock::from_utc(t) is well-formed.
5. Mandates: DestClock::from_utc(t) returns a time_point<DestClock, Duration>, where Duration
is a valid chrono::duration specialization.
6. Returns: DestClock::from_utc(t).

27.7.10.6 Function template clock_cast

```cpp
template<class DestClock, class SourceClock, class Duration>
auto clock_cast(const time_point<SourceClock, Duration>& t);
```

1. Constraints: At least one of the following clock time conversion expressions is well-formed:
   (1.1) clock_time_conversion<DestClock, SourceClock>{t}
   (1.2) clock_time_conversion<DestClock, system_clock>{
         clock_time_conversion<system_clock, SourceClock>{t}}
   (1.3) clock_time_conversion<DestClock, utc_clock>{
         clock_time_conversion<utc_clock, SourceClock>{t}}
   (1.4) clock_time_conversion<DestClock, utc_clock>{
         clock_time_conversion<utc_clock, system_clock>{
           clock_time_conversion<system_clock, SourceClock>{t}}}
   (1.5) clock_time_conversion<DestClock, system_clock>{
         clock_time_conversion<system_clock, utc_clock>{
           clock_time_conversion<utc_clock, SourceClock>{t}}}

A clock time conversion expression is considered better than another clock time conversion expression
if it involves fewer operator() calls on clock_time_conversion specializations.
2. Mandates: Among the well-formed clock time conversion expressions from the above list, there is a
unique best expression.
3. Returns: The best well-formed clock time conversion expression in the above list.

27.8 The civil calendar

27.8.1 In general

1 The types in 27.8 describe the civil (Gregorian) calendar and its relationship to sys_days and local_days.
27.8.2 Class last_spec

namespace std::chrono {
  struct last_spec {
    explicit last_spec() = default;
  };
}

The type last_spec is used in conjunction with other calendar types to specify the last in a sequence. For example, depending on context, it can represent the last day of a month, or the last day of the week of a month.

27.8.3 Class day

27.8.3.1 Overview

namespace std::chrono {
  class day {
    unsigned char d_;  // exposition only
  public:
    day() = default;
    constexpr explicit day(unsigned d) noexcept;
   constexpr day& operator++() noexcept;
    constexpr day operator++(int) noexcept;
    constexpr day& operator--() noexcept;
    constexpr day operator--(int) noexcept;
    constexpr day& operator+=(const days& d) noexcept;
    constexpr day& operator-=(const days& d) noexcept;
    constexpr explicit operator unsigned() const noexcept;
    constexpr bool ok() const noexcept;
  };
}

day represents a day of a month. It normally holds values in the range 1 to 31, but may hold non-negative values outside this range. It can be constructed with any unsigned value, which will be subsequently truncated to fit into day’s unspecified internal storage. day meets the Cpp17EqualityComparable (Table 25) and Cpp17LessThanComparable (Table 26) requirements, and participates in basic arithmetic with days objects, which represent a difference between two day objects.

day is a trivially copyable and standard-layout class type.

27.8.3.2 Member functions

constexpr explicit day(unsigned d) noexcept;

1 Effects: Initializes d with d. The value held is unspecified if d is not in the range [0, 255].

constexpr day& operator++() noexcept;

2 Effects: ++d.

3 Returns: *this.

constexpr day operator++(int) noexcept;

4 Effects: ++(*this).

5 Returns: A copy of *this as it existed on entry to this member function.

constexpr day& operator--() noexcept;

6 Effects: Equivalent to: --d.

7 Returns: *this.

constexpr day operator--(int) noexcept;

8 Effects: --(*this).

9 Returns: A copy of *this as it existed on entry to this member function.
constexpr day& operator+=(const days& d) noexcept;

* Effects: *this = *this + d.
* Returns: *this.

constexpr day& operator-=(const days& d) noexcept;

* Effects: *this = *this - d.
* Returns: *this.

constexpr explicit operator unsigned() const noexcept;

* Returns: d_.

constexpr bool ok() const noexcept;

* Returns: 1 <= d_ && d_ <= 31.

27.8.3.3 Non-member functions

constexpr bool operator==(const day& x, const day& y) noexcept;

* Returns: unsigned{x} == unsigned{y}.

constexpr strong_ordering operator<=>(const day& x, const day& y) noexcept;

* Returns: unsigned{x} <=> unsigned{y}.

constexpr day operator+(const day& x, const days& y) noexcept;

* Returns: y + x.

constexpr day operator+(const days& x, const day& y) noexcept;

* Returns: day(unsigned{x} + y.count()).

constexpr day operator-(const day& x, const days& y) noexcept;

* Returns: x + -y.

constexpr days operator-(const day& x, const day& y) noexcept;

* Returns: days{int(unsigned{x}) - int(unsigned{y})}.

template<class charT, class traits>

basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const day& d);

* Effects: Equivalent to:

    return os << (d.ok() ?
        format(STATICALLY-WIDEN<charT>("{%d}"), d) :
        format(STATICALLY-WIDEN<charT>("{%d} is not a valid day"), d));

template<class charT, class traits, class Alloc = allocator<charT>>

basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt, day& d, basic_string<charT, traits, Alloc>* abbrev = nullptr, minutes* offset = nullptr);

* Effects: Attempts to parse the input stream is into the day d using the format flags given in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid day, is.setstate(ios_base::failbit) is called and d is not modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null.

* Returns: is.

constexpr chrono::day operator""d(unsigned long long d) noexcept;

* Returns: day{static_cast<unsigned>(d)}.
27.8.4 Class month

27.8.4.1 Overview

namespace std::chrono {
    class month {
        unsigned char m_;  // exposition only
    public:
        month() = default;
        constexpr explicit month(unsigned m) noexcept;
        constexpr month& operator++() noexcept;
        constexpr month operator++(int) noexcept;
        constexpr month& operator--() noexcept;
        constexpr month operator--(int) noexcept;
        constexpr month& operator+=(const months& m) noexcept;
        constexpr month& operator-=(const months& m) noexcept;
        constexpr explicit operator unsigned() const noexcept;
        constexpr bool ok() const noexcept;
    };
}

1 *month* represents a month of a year. It normally holds values in the range 1 to 12, but may hold non-negative values outside this range. It can be constructed with any *unsigned* value, which will be subsequently truncated to fit into *month*'s unspecified internal storage. *month* meets the Cpp17EqualityComparable (Table 25) and Cpp17LessThanComparable (Table 26) requirements, and participates in basic arithmetic with *months* objects, which represent a difference between two *month* objects.

2 *month* is a trivially copyable and standard-layout class type.

27.8.4.2 Member functions

constexpr explicit month(unsigned m) noexcept;

1 *Effects*: Initializes *m* with *m*. The value held is unspecified if *m* is not in the range [0, 255].

constexpr month& operator++() noexcept;

2 *Effects*: *this += months(1).

3 *Returns*: *this.

constexpr month operator++(int) noexcept;

4 *Effects*: ++(*this).

5 *Returns*: A copy of *this* as it existed on entry to this member function.

constexpr month& operator--() noexcept;

6 *Effects*: *this -= months(1).

7 *Returns*: *this.

constexpr month operator--(int) noexcept;

8 *Effects*: --(*this).

9 *Returns*: A copy of *this* as it existed on entry to this member function.

constexpr month& operator+=(const months& m) noexcept;

10 *Effects*: *this = *this + m.

11 *Returns*: *this.

constexpr month& operator-=(const months& m) noexcept;

12 *Effects*: *this = *this - m.

13 *Returns*: *this.
constexpr explicit operator unsigned() const noexcept;

Returns: \( m_\).  

constexpr bool ok() const noexcept;

Returns: \( 1 \leq m_ \land m_ \leq 12 \).

27.8.4.3 Non-member functions

constexpr bool operator==(const month& x, const month& y) noexcept;

Returns: \( \text{unsigned}\{x\} == \text{unsigned}\{y\} \).

constexpr strong_ordering operator<=>(const month& x, const month& y) noexcept;

Returns: \( \text{unsigned}\{x\} \langle => \text{unsigned}\{y\} \).

constexpr month operator+(const month& x, const months& y) noexcept;

Returns:
\[
\text{month}\left(\text{modulo}(\text{static\_cast}\langle\text{long\_long}\rangle(\text{unsigned}\{x\}) + (y.\text{count}() - 1), 12) + 1\right)
\]
where \( \text{modulo}(n, 12) \) computes the remainder of \( n \) divided by 12 using Euclidean division.

[Note 1: Given a divisor of 12, Euclidean division truncates towards negative infinity and always produces a remainder in the range of \([0, 11]\). Assuming no overflow in the signed summation, this operation results in a month holding a value in the range \([1, 12]\) even if \( !x.\text{ok}() \). —end note]

[Example 1: February + months\{11\} == January. —end example]

constexpr month operator+(const months& x, const month& y) noexcept;

Returns: \( y + x \).

constexpr month operator-(const month& x, const months& y) noexcept;

Returns: \( x + -y \).

constexpr months operator-(const month& x, const month& y) noexcept;

Returns: If \( x.\text{ok}() == \text{true} \) and \( y.\text{ok}() == \text{true} \), returns a value \( m \) in the range \([\text{months}\{0\}, \text{months}\{11\}]\) satisfying \( y + m == x \). Otherwise the value returned is unspecified.

[Example 2: January - February == months\{11\}. —end example]

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month& m);

Effects: Equivalent to:
\[
\text{return } os << \left( m.\text{ok}() ? \right. \\
\text{format}(os.\text{getloc}(), \text{STATICALLY-WIDEN}<\text{charT}>\langle\{:%b\}\rangle, m) : \\
\text{format}(os.\text{getloc}(), \text{STATICALLY-WIDEN}<\text{charT}>\langle\{\} is not a valid month\rangle, \right. \\
\text{static\_cast}<\text{unsigned}\langle m\rangle));
\]

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt, 
month& m, basic_string<charT, traits, Alloc>** abbrev = nullptr, 
minutes* offset = nullptr);

Effects: Attempts to parse the input stream \( is \) into the month \( m \) using the format flags given in the NTCTS \( \text{fmt} \) as specified in 27.13. If the parse fails to decode a valid month, \( is.\text{setstate}(\text{ios\_base::failbit}) \) is called and \( m \) is not modified. If \%Z is used and successfully parsed, that value will be assigned to \*abbrev if \abbrev\ is non-null. If \%z (or a modified variant) is used and successfully parsed, that value will be assigned to \*offset if \offset\ is non-null.

Returns: \( is \).
27.8.5 Class year

27.8.5.1 Overview

namespace std::chrono {
    class year {
        short y_;  // exposition only
    public:
        year() = default;
        constexpr explicit year(int y) noexcept;
        constexpr year& operator++() noexcept;
        constexpr year operator++(int) noexcept;
        constexpr year& operator--() noexcept;
        constexpr year operator--(int) noexcept;
        constexpr year& operator+=(const years& y) noexcept;
        constexpr year& operator-=(const years& y) noexcept;
        constexpr year operator+() const noexcept;
        constexpr year operator-() const noexcept;
        constexpr bool is_leap() const noexcept;
        constexpr explicit operator int() const noexcept;
        constexpr bool ok() const noexcept;
        static constexpr year min() noexcept;
        static constexpr year max() noexcept;
    };
}

1 year represents a year in the civil calendar. It can represent values in the range \([\text{min}(), \text{max}()]\). It can be constructed with any int value, which will be subsequently truncated to fit into year’s unspecified internal storage. year meets the Cpp17EqualityComparable (Table 25) and Cpp17LessThanComparable (Table 26) requirements, and participates in basic arithmetic with years objects, which represent a difference between two year objects.

2 year is a trivially copyable and standard-layout class type.

27.8.5.2 Member functions

constexpr explicit year(int y) noexcept;

1 Effects: Initializes y_ with y. The value held is unspecified if y is not in the range \([-32767, 32767]\).

constexpr year& operator++() noexcept;

2 Effects: ++y_.
3 Returns: *this.

constexpr year operator++(int) noexcept;

4 Effects: ++(*this).
5 Returns: A copy of *this as it existed on entry to this member function.

constexpr year& operator--() noexcept;

6 Effects: --y_.
7 Returns: *this.

constexpr year operator--(int) noexcept;

8 Effects: --(*this).
9 Returns: A copy of *this as it existed on entry to this member function.
constexpr year& operator+=(const years& y) noexcept;
  Effects: *this = *this + y.
  Returns: *this.

constexpr year& operator-=(const years& y) noexcept;
  Effects: *this = *this - y.
  Returns: *this.

constexpr year operator+() const noexcept;
  Returns: *this.

constexpr year operator-() const noexcept;
  Returns: year{-y_}.

constexpr bool is_leap() const noexcept;
  Returns: y_ % 4 == 0 && (y_ % 100 != 0 || y_ % 400 == 0).

constexpr explicit operator int() const noexcept;
  Returns: y_.

constexpr bool ok() const noexcept;
  Returns: min().y_ <= y_ && y_ <= max().y_.

static constexpr year min() noexcept;
  Returns: year{-32767}.

static constexpr year max() noexcept;
  Returns: year{32767}.

27.8.5.3 Non-member functions

constexpr bool operator==(const year& x, const year& y) noexcept;
  Returns: int{x} == int{y}.

constexpr strong_ordering operator<=>(const year& x, const year& y) noexcept;
  Returns: int{x} <=> int{y}.

constexpr year operator+(const year& x, const years& y) noexcept;
  Returns: year{int{x} + y.count()}.

constexpr year operator+(const years& x, const year& y) noexcept;
  Returns: y + x.

constexpr year operator-(const year& x, const years& y) noexcept;
  Returns: x + -y.

constexpr years operator-(const year& x, const year& y) noexcept;
  Returns: years{int{x} - int{y}}.

template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const year& y);
  Effects: Equivalent to:
    return os << (y.ok() ?
      format(STATICALLY-WIDEN<charT>("{:Y}\n"), y) :
      format(STATICALLY-WIDEN<charT>("{:Y} is not a valid year"), y));
template<class charT, class traits, class Alloc = allocator<charT>>
  basic_istream<charT, traits>& from_stream(basic_istream<charT, traits>& is, const charT* fmt,
  year& y, basic_string<charT, traits, Alloc>* abbrev = nullptr,
  minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the year y using the format flags given in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid year, is.setstate(ios_base::failbit) is called and y is not modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null.

Returns: is.

constexpr chrono::year operator"y(unsigned long long y) noexcept;

Returns: year(static_cast<int>(y)).

27.8.6 Class weekday

27.8.6.1 Overview

namespace std::chrono {
  class weekday {
    unsigned char wd_; // exposition only
  public:
    weekday() = default;
  constexpr explicit weekday(unsigned wd) noexcept;
  constexpr weekday(const sys_days& dp) noexcept;
  constexpr explicit weekday(const local_days& dp) noexcept;

    constexpr weekday& operator++() noexcept;
    constexpr weekday operator++(int) noexcept;
    constexpr weekday& operator--() noexcept;
    constexpr weekday operator--(int) noexcept;

    constexpr weekday& operator+=(const days& d) noexcept;
    constexpr weekday& operator-=(const days& d) noexcept;

    constexpr unsigned c_encoding() const noexcept;
    constexpr unsigned iso_encoding() const noexcept;
    constexpr bool ok() const noexcept;

    constexpr weekday_indexed operator[](unsigned index) const noexcept;
    constexpr weekday_last operator[](last_spec) const noexcept;
  };
};

weekday represents a day of the week in the civil calendar. It normally holds values in the range 0 to 6, corresponding to Sunday through Saturday, but it may hold non-negative values outside this range. It can be constructed with any unsigned value, which will be subsequently truncated to fit into weekday's unspecified internal storage. weekday meets the Cpp17EqualityComparable (Table 25) requirements.

[Note 1: weekday is not Cpp17LessThanComparable because there is no universal consensus on which day is the first day of the week. weekday's arithmetic operations treat the days of the week as a circular range, with no beginning and no end. — end note]

weekday is a trivially copyable and standard-layout class type.

27.8.6.2 Member functions

constexpr explicit weekday(unsigned wd) noexcept;

Effects: Initializes wd_ with wd == 7 ? 0 : wd. The value held is unspecified if wd is not in the range [0, 255].

constexpr weekday(const sys_days& dp) noexcept;

Effects: Computes what day of the week corresponds to the sys_days dp, and initializes that day of the week in wd_.

§ 27.8.6.2
constexpr explicit weekday(const local_days& dp) noexcept;

Effects: Computes what day of the week corresponds to the local_days dp, and initializes that day of the week in wd_.
Postconditions: The value is identical to that constructed from sys_days{dp.time_since_epoch()}.

constexpr weekday& operator++() noexcept;

Effects: *this += days(1).
Returns: *this.

constexpr weekday& operator++(int) noexcept;

Effects: ++(*this).
Returns: A copy of *this as it existed on entry to this member function.

constexpr weekday& operator--() noexcept;

Effects: *this -= days(1).
Returns: *this.

constexpr weekday operator--(int) noexcept;

Effects: --(*this).
Returns: A copy of *this as it existed on entry to this member function.

constexpr weekday& operator+=(const days& d) noexcept;

Effects: *this = *this + d.
Returns: *this.

constexpr weekday& operator-=(const days& d) noexcept;

Effects: *this = *this - d.
Returns: *this.

constexpr unsigned c_encoding() const noexcept;

Returns: wd_.

constexpr unsigned iso_encoding() const noexcept;

Returns: wd_ == 0u ? 7u : wd_.

constexpr bool ok() const noexcept;

Returns: wd_ <= 6.

constexpr weekday_indexed operator[](unsigned index) const noexcept;

Returns: {*this, index}.

constexpr weekday_last operator[](last_spec) const noexcept;

Returns: weekday_last{*this}.

27.8.6.3 Non-member functions [time.cal.wd.nonmembers]

constexpr bool operator==(const weekday& x, const weekday& y) noexcept;

Returns: x.wd_ == y.wd_.

constexpr weekday operator+(const weekday& x, const days& y) noexcept;

Returns:
weekday{modulo(static_cast<long long>(x.wd_) + y.count(), 7)}
where modulo(n, 7) computes the remainder of n divided by 7 using Euclidean division.
[Note 1: Given a divisor of 7, Euclidean division truncates towards negative infinity and always produces a remainder in the range of \([0, 6]\). Assuming no overflow in the signed summation, this operation results in a \textit{weekday} holding a value in the range \([0, 6]\) even if \(!\text{x.ok}().\) — end note]

\begin{itemize}
\item constexpr weekday operator+(const days& x, const weekday& y) noexcept;
\end{itemize}

\textit{Returns:} \(y + x\).

\begin{itemize}
\item constexpr weekday operator-(const weekday& x, const days& y) noexcept;
\end{itemize}

\textit{Returns:} \(x - y\).

\begin{itemize}
\item constexpr days operator-(const weekday& x, const weekday& y) noexcept;
\end{itemize}

\textit{Returns:} If \(x.ok() == \text{true}\) and \(y.ok() == \text{true}\), returns a value \(d\) in the range \([\text{days}\{0\}, \text{days}\{6\}]\) satisfying \(y + d == x\). Otherwise the value returned is unspecified.

\begin{itemize}
\item Example 1: Monday + days\{6\} == Sunday. — end example
\end{itemize}

\begin{itemize}
\item template<class charT, class traits>
\item basic_ostream\langle charT, traits\rangle&
\item operator<<(basic_ostream\langle charT, traits\rangle& os, const weekday& wd);
\end{itemize}

\textit{Effects:} Equivalent to:

\begin{itemize}
\item return os << (wd.ok() ?
\item format(os.getloc(), STATICALLY-WIDEN\langle charT\rangle("{:\%a}\"), wd) :
\item format(os.getloc(), STATICALLY-WIDEN\langle charT\rangle("\{\} is not a valid weekday"),
\item static_cast\langle unsigned\rangle\langle\text{wd.wd}_1\rangle);
\end{itemize}

\begin{itemize}
\item template<class charT, class traits, class Alloc = allocator\langle charT\rangle>
\item basic_istream\langle charT, traits\rangle&
\item from_stream(basic_istream\langle charT, traits\rangle& is, const charT* fmt, 
\item weekday& wd, basic_string\langle charT, traits, Alloc\rangle* abbrev = nullptr, 
\item minutes* offset = nullptr);
\end{itemize}

\textit{Effects:} Attempts to parse the input stream \(\text{is}\) into the \textit{weekday} \(\text{wd}\) using the format flags given in the NTCTS \(\text{fmt}\) as specified in 27.13. If the parse fails to decode a valid weekday, \(\text{is.setstate(ios_-
\item base::failbit)}\) is called and \(\text{wd}\) is not modified. If \%Z is used and successfully parsed, that value will be assigned to \(*\text{abbrev}\) if \(\text{abbrev}\) is non-null. If \%z (or a modified variant) is used and successfully parsed, that value will be assigned to \(*\text{offset}\) if \(\text{offset}\) is non-null.

\textit{Returns:} \(\text{is}\).

27.8.7 Class \texttt{weekday\_indexed}

27.8.7.1 Overview

\begin{itemize}
\item namespace std::chrono {
\item class \texttt{weekday\_indexed} {
\item \texttt{chrono::weekday \_wd;} \hfill // exposition only
\item unsigned char \_index; \hfill // exposition only
\item 
\item public:
\item weekday\_indexed() = default;
\item constexpr weekday\_indexed(const chrono::weekday& \_wd, unsigned \_index) noexcept;
\item \texttt{constexpr} chrono::weekday weekday() const noexcept;
\item constexpr unsigned index() const noexcept;
\item constexpr bool ok() const noexcept;
\item 
\item }
\item }
\end{itemize}

\texttt{weekday\_indexed} represents a \textit{weekday} and a small index in the range 1 to 5. This class is used to represent the first, second, third, fourth, or fifth weekday of a month.

\begin{itemize}
\item [Note 1: A \texttt{weekday\_indexed} object can be constructed by indexing a \textit{weekday} with an \texttt{unsigned}. — end note]
\item [Example 1:]
\item constexpr auto \texttt{wdi} = Sunday\{2\}; // \texttt{wdi} is the second Sunday of an as yet unspecified month
\end{itemize}
static_assert(wdi.weekday() == Sunday);
static_assert(wdi.index() == 2);

—end example]

weekday_indexed is a trivially copyable and standard-layout class type.

27.8.7.2 Member functions

constexpr weekday_indexed(const chrono::weekday& wd, unsigned index) noexcept;

Effects: Initializes wd_ with wd and index_ with index. The values held are unspecified if !wd.ok() or index is not in the range [0, 7].

constexpr chrono::weekday weekday() const noexcept;

Returns: wd_.

constexpr unsigned index() const noexcept;

Returns: index_.

constexpr bool ok() const noexcept;

Returns: wd_.ok() && 1 <= index_ && index_ <= 5.

27.8.7.3 Non-member functions

constexpr bool operator==(const weekday_indexed& x, const weekday_indexed& y) noexcept;

Returns: x.weekday() == y.weekday() && x.index() == y.index().

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const weekday_indexed& wdi);

Effects: Equivalent to:

auto i = wdi.index();
return os << (i >= 1 && i <= 5 ?
    format(os.getloc(), STATICALLY-WIDEN<charT>("{}[{}]"), wdi.weekday(), i) :
    format(os.getloc(), STATICALLY-WIDEN<charT>("{}[{} is not a valid index]"),
        wdi.weekday(), i));

27.8.8 Class weekday_last

27.8.8.1 Overview

namespace std::chrono {
    class weekday_last {
    chronos::weekday wd_; // exposition only

    public:
    constexpr explicit weekday_last(const chrono::weekday& wd) noexcept;

    constexpr chrono::weekday weekday() const noexcept;
    constexpr bool ok() const noexcept;
    }
}

weekday_last represents the last weekday of a month.

[Note 1: A weekday_last object can be constructed by indexing a weekday with last. — end note]

[Example 1:

constexpr auto wdl = Sunday[last];  // wdl is the last Sunday of an as yet unspecified month
static_assert(wdl.weekday() == Sunday);

—end example]

weekday_last is a trivially copyable and standard-layout class type.
27.8.8.2 Member functions

```cpp
constexpr explicit weekday_last(const chrono::weekday& wd) noexcept;
```

**Effects:** Initializes \( wd_ \) with \( wd \).

**Returns:** \( wd_ \).

```cpp
constexpr chrono::weekday weekday() const noexcept;
```

**Returns:** \( wd_ \).

```cpp
constexpr bool ok() const noexcept;
```

**Returns:** \( \text{true} \) if \( m_.\text{ok}() \text{ is } \text{true} \), \( 1 \text{d} \leq d_ \), and \( d_ \) is less than or equal to the number of days in month \( m_ \); otherwise returns \( \text{false} \). When \( m_ \text{ == February} \), the number of days is considered to be 29.

27.8.8.3 Non-member functions

```cpp
constexpr bool operator==(const weekday_last& x, const weekday_last& y) noexcept;
```

**Returns:** \( x.\text{weekday}() == y.\text{weekday}() \).

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const weekday_last& wdl);
```

**Effects:** Equivalent to:

\[
\text{return os} \ll \text{format(os.getloc(), \text{STATICALLY-WIDEN}<charT>("{}[last]"), wdl.\text{weekday})};
\]

27.8.9 Class `month_day`

27.8.9.1 Overview

```cpp
namespace std::chrono {
    class month_day {
        chrono::month m_; // exposition only
        chrono::day d_; // exposition only

        public:
            month_day() = default;
            constexpr month_day(const chrono::month& m, const chrono::day& d) noexcept;

            constexpr chrono::month month() const noexcept;
            constexpr chrono::day day() const noexcept;
            constexpr bool ok() const noexcept;
    };
}
```

`month_day` represents a specific day of a specific month, but with an unspecified year. `month_day` meets the `Cpp17EqualityComparable` (Table 25) and `Cpp17LessThanComparable` (Table 26) requirements.

`month_day` is a trivially copyable and standard-layout class type.

27.8.9.2 Member functions

```cpp
constexpr month_day(const chrono::month& m, const chrono::day& d) noexcept;
```

**Effects:** Initializes \( m_ \) with \( m \), and \( d_ \) with \( d \).

**Returns:** \( m_ \).

```cpp
constexpr chrono::month month() const noexcept;
```

**Returns:** \( m_ \).

```cpp
constexpr chrono::day day() const noexcept;
```

**Returns:** \( d_ \).

```cpp
constexpr bool ok() const noexcept;
```

**Returns:** \( \text{true} \) if \( m_.\text{ok}() \text{ is } \text{true} \), \( 1 \text{d} \leq d_ \), and \( d_ \) is less than or equal to the number of days in month \( m_ \); otherwise returns \( \text{false} \). When \( m_ \text{ == February} \), the number of days is considered to be 29.
Non-member functions

constexpr bool operator==(const month_day& x, const month_day& y) noexcept;

Returns: x.month() == y.month() && x.day() == y.day().

constexpr strong_ordering operator<=>(const month_day& x, const month_day& y) noexcept;

Effects: Equivalent to:

if (auto c = x.month() <=> y.month(); c != 0) return c;
return x.day() <=> y.day();

template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, const month_day& md);

Effects: Equivalent to:

return os << format(os.getloc(), STATICALLY-WIDEN<charT>("/{}/{}”),
md.month(), md.day());

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
month_day& md, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the month_day md using the format flags given in the
NTCTS fmt as specified in 27.13. If the parse fails to decode a valid month_day, is.setstate(ios_base::failbit)
is called and md is not modified. If %Z is used and successfully parsed, that value will
be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully
parsed, that value will be assigned to *offset if offset is non-null.

Returns: is.

Class month_day_last

namespace std::chrono {
  class month_day_last {
  // exposition only
  
    public:
    constexpr explicit month_day_last(const chrono::month& m) noexcept;
    
    constexpr chrono::month month() const noexcept;
    constexpr bool ok() const noexcept;
  
  };
}

month_day_last represents the last day of a month.

[Note 1: A month_day_last object can be constructed using the expression m/last or last/m, where m is an expression
of type month. —end note]

[Example 1:]

constexpr auto mdl = February/last; // mdl is the last day of February of an as yet unspecified year
static_assert(mdl.month() == February);

—end example]

month_day_last is a trivially copyable and standard-layout class type.

constexpr explicit month_day_last(const chrono::month& m) noexcept;

Effects: Initializes m_ with m.

constexpr month month() const noexcept;

Returns: m_.

constexpr bool ok() const noexcept;

Returns: m_.ok().
constexpr bool operator==(const month_day_last& x, const month_day_last& y) noexcept;

Returns: \texttt{x.month()} == \texttt{y.month()}.

constexpr strong_ordering operator<=>(const month_day_last& x, const month_day_last& y) noexcept;

Returns: \texttt{x.month()} <=> \texttt{y.month()}.

template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const month_day_last& mdl);

Effects: Equivalent to:

\[
    \text{return os << format(os.getloc(), \texttt{STATICALLY-WIDEN}<charT>("\{}/last"), mdl.month());}
\]

27.8.11 Class \texttt{month\_weekday} \[time.cal.mwd\]

27.8.11.1 Overview \[time.cal.mwd.overview\]

namespace std::chrono {
    class month_weekday {
        chrono::month m_; // exposition only
        chrono::weekday_indexed wdi_; // exposition only
    public:
        constexpr month_weekday(const chrono::month& m, const chrono::weekday_indexed& wdi) noexcept;
        constexpr chrono::month month() const noexcept;
        constexpr chrono::weekday_indexed weekday_indexed() const noexcept;
        constexpr bool ok() const noexcept;
    }
}

\texttt{month\_weekday} represents the \(n\)th weekday of a month, of an as yet unspecified year. To do this the \texttt{month\_weekday} stores a \texttt{month} and a \texttt{weekday\_indexed}.

\textbf{Example 1:}

\begin{verbatim}
constexpr auto mwd
    = February/Tuesday[3]; // mwd is the third Tuesday of February of an as yet unspecified year
static_assert(mwd.month() == February);
static_assert(mwd.weekday_indexed() == Tuesday[3]);
\end{verbatim}

\texttt{month\_weekday} is a trivially copyable and standard-layout class type.

27.8.11.2 Member functions \[time.cal.mwd.members\]

constexpr month_weekday(const chrono::month& m, const chrono::weekday_indexed& wdi) noexcept;

Effects: Initializes \texttt{m\_} with \texttt{m}, and \texttt{wdi\_} with \texttt{wdi}.

constexpr chrono::month month() const noexcept;

Returns: \texttt{m\_}.

constexpr chrono::weekday_indexed weekday_indexed() const noexcept;

Returns: \texttt{wdi\_}.

constexpr bool ok() const noexcept;

Returns: \texttt{m\_}.\texttt{ok()} \&\& \texttt{wdi\_.ok()}.

27.8.11.3 Non-member functions \[time.cal.mwd.nonmembers\]

constexpr bool operator==(const month_weekday& x, const month_weekday& y) noexcept;

Returns: \texttt{x.month()} == \texttt{y.month()} \&\& \texttt{x.weekday\_indexed()} == \texttt{y.weekday\_indexed()}.

template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const month_weekday& mdl);

Effects: Equivalent to:
return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}/{}"),
    mwd.month(), mwd.weekday_indexed());

27.8.12 Class month_weekday_last

27.8.12.1 Overview

namespace std::chrono {
    class month_weekday_last {
        chrono::month m_;  // exposition only
        chrono::weekday_last wdl_;  // exposition only
        public:
            constexpr month_weekday_last(const chrono::month& m,
                const chrono::weekday_last& wdl) noexcept;
            constexpr chrono::month month() const noexcept;
            constexpr chrono::weekday_last weekday_last() const noexcept;
            constexpr bool ok() const noexcept;
    };
}

1 month_weekday_last represents the last weekday of a month, of an as yet unspecified year. To do this the
month_weekday_last stores a month and a weekday_last.

2 [Example 1]:
    constexpr auto mwd = February/Tuesday[last];  // mwd is the last Tuesday of February of an as yet unspecified year
    static_assert(mwd.month() == February);
    static_assert(mwd.weekday_last() == Tuesday[last]);
    —end example

3 month_weekday_last is a trivially copyable and standard-layout class type.

27.8.12.2 Member functions

caseexpr month_weekday_last(const chrono::month& m,
    const chrono::weekday_last& wdl) noexcept;
1    Effects: Initializes m_ with m, and wdl_ with wdl.

constexpr chrono::month month() const noexcept;
2    Returns: m_.

constexpr chrono::weekday_last weekday_last() const noexcept;
3    Returns: wdl_.

constexpr bool ok() const noexcept;
4    Returns: m_.ok() && wdl_.ok().

27.8.12.3 Non-member functions

caseexpr bool operator==(const month_weekday_last& x, const month_weekday_last& y) noexcept;
1    Returns: x.month() == y.month() && x.weekday_last() == y.weekday_last().

template<class charT, class traits>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const month_weekday_last& mwd);
2    Effects: Equivalent to:
        return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}/{}"),
            mwd.month(), mwd.weekday_last());
27.8.13 Class `year_month`  

27.8.13.1 Overview

```cpp
namespace std::chrono {
    class year_month {
        chrono::year y_; // exposition only
        chrono::month m_; // exposition only

    public:
        year_month() = default;
        constexpr year_month(const chrono::year& y, const chrono::month& m) noexcept;

        constexpr chrono::year year() const noexcept;
        constexpr chrono::month month() const noexcept;

        constexpr year_month& operator+=(const months& dm) noexcept;
        constexpr year_month& operator-=(const months& dm) noexcept;

        constexpr year_month& operator+=(const years& dy) noexcept;
        constexpr year_month& operator-=(const years& dy) noexcept;

        constexpr bool ok() const noexcept;
    }
}
```

`year_month` represents a specific month of a specific year, but with an unspecified day. `year_month` is a field-based time point with a resolution of months. `year_month` meets the Cpp17EqualityComparable (Table 25) and Cpp17LessThanComparable (Table 26) requirements.

`year_month` is a trivially copyable and standard-layout class type.

27.8.13.2 Member functions

```cpp
constexpr year_month(const chrono::year& y, const chrono::month& m) noexcept;
```

Effects: Initializes `y_` with `y`, and `m_` with `m`.

Returns: `y_`.

```cpp
constexpr chrono::year year() const noexcept;
```

Returns: `y_`.

```cpp
constexpr chrono::month month() const noexcept;
```

Returns: `m_`.

```cpp
constexpr year_month& operator+=(const months& dm) noexcept;
```

Constraints: If the argument supplied by the caller for the `months` parameter is convertible to `years`, its implicit conversion sequence to `years` is worse than its implicit conversion sequence to `months` (12.4.4.3).

Effects: `*this = *this + dm`.

Returns: `*this`.

```cpp
constexpr year_month& operator-=(const months& dm) noexcept;
```

Constraints: If the argument supplied by the caller for the `months` parameter is convertible to `years`, its implicit conversion sequence to `years` is worse than its implicit conversion sequence to `months` (12.4.4.3).

Effects: `*this = *this - dm`.

Returns: `*this`.

```cpp
constexpr year_month& operator+=(const years& dy) noexcept;
```

Effects: `*this = *this + dy`.

Returns: `*this`.

```cpp
constexpr year_month& operator-=(const years& dy) noexcept;
```

Effects: `*this = *this - dy`.

Returns: `*this`.

§ 27.8.13.2
constexpr bool ok() const noexcept;
Returns: y_.ok() & m_.ok().

27.8.13.3 Non-member functions [time.cal.ym.nonmembers]

constexpr bool operator==(const year_month& x, const year_month& y) noexcept;
Returns: x.year() == y.year() && x.month() == y.month().

constexpr strong_ordering operator<=>(const year_month& x, const year_month& y) noexcept;
Effects: Equivalent to:
if (auto c = x.year() <=> y.year(); c != 0) return c;
return x.month() <=> y.month();

constexpr year_month operator+(const year_month& ym, const months& dm) noexcept;
Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).
Returns: A year_month value z such that z.ok() && z - ym == dm is true.
Complexity: \( \Theta(1) \) with respect to the value of dm.

constexpr year_month operator+(const months& dm, const year_month& ym) noexcept;
Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).
Returns: ym + dm.

constexpr year_month operator-(const year_month& ym, const months& dm) noexcept;
Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).
Returns: ym + -dm.

constexpr months operator-(const year_month& x, const year_month& y) noexcept;
Returns:
x.year() - y.year() + months{static_cast<int>(unsigned{x.month()}) -
static_cast<int>(unsigned{y.month()})}

constexpr year_month operator+(const year_month& ym, const years& dy) noexcept;
Returns: (ym.year() + dy) / ym.month().

constexpr year_month operator+(const years& dy, const year_month& ym) noexcept;
Returns: ym + dy.

constexpr year_month operator-(const year_month& ym, const years& dy) noexcept;
Returns: ym + -dy.

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month& ym);
Effects: Equivalent to:
return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}/{}"),
ym.year(), ym.month());

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, charT* fmt,
year_month& ym, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);
Effects: Attempts to parse the input stream is into the year_month ym using the format flags given in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid year_month, is.setstate(ios-
base::failbit) is called and ym is not modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null.

Returns: is.

27.8.14 Class year_month_day

27.8.14.1 Overview

namespace std::chrono {
    class year_month_day {
        chrono::year y_; // exposition only
        chrono::month m_; // exposition only
        chrono::day d_; // exposition only

    public:
        year_month_day() = default;
        constexpr year_month_day(const chrono::year& y, const chrono::month& m,
                                  const chrono::day& d) noexcept;
        constexpr year_month_day(const year_month_day_last& ymdl) noexcept;
        constexpr year_month_day(const sys_days& dp) noexcept;
        constexpr explicit year_month_day(const local_days& dp) noexcept;
        constexpr year_month_day& operator+=(const months& m) noexcept;
        constexpr year_month_day& operator-=(const months& m) noexcept;
        constexpr year_month_day& operator+=(const years& y) noexcept;
        constexpr year_month_day& operator-=(const years& y) noexcept;
        constexpr chrono::year year() const noexcept;
        constexpr chrono::month month() const noexcept;
        constexpr chrono::day day() const noexcept;
        constexpr operator sys_days() const noexcept;
        constexpr explicit operator local_days() const noexcept;
        constexpr bool ok() const noexcept;
    };
}

1 year_month_day represents a specific year, month, and day. year_month_day is a field-based time point with a resolution of days.

[Note 1: year_month_day supports years- and months-oriented arithmetic, but not days-oriented arithmetic. For the latter, there is a conversion to sys_days, which efficiently supports days-oriented arithmetic. — end note]

year_month_day meets the Cpp17EqualityComparable (Table 25) and Cpp17LessThanComparable (Table 26) requirements.

2 year_month_day is a trivially copyable and standard-layout class type.

27.8.14.2 Member functions

constexpr year_month_day(const chrono::year& y, const chrono::month& m,
                         const chrono::day& d) noexcept;

1 Effects: Initializes y_ with y, m_ with m, and d_ with d.

constexpr year_month_day(const year_month_day_last& ymdl) noexcept;

2 Effects: Initializes y_ with ymdl.year(), m_ with ymdl.month(), and d_ with ymdl.day().

[Note 1: This conversion from year_month_day_last to year_month_day might be more efficient than converting a year_month_day_last to a sys_days, and then converting that sys_days to a year_month_day. — end note]

constexpr year_month_day(const sys_days& dp) noexcept;

3 Effects: Constructs an object of type year_month_day that corresponds to the date represented by dp.

Remarks: For any value ymd of type year_month_day for which ymd.ok() is true, ymd == year_-month_day(sys_days{ymd}) is true.
constexpr explicit year_month_day(const local_days& dp) noexcept;

Effects: Equivalent to constructing with sys_days{dp.time_since_epoch()}.

constexpr year_month_day& operator+=(const months& m) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Effects: *this = *this + m.

Returns: *this.

constexpr year_month_day& operator-=(const months& m) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Effects: *this = *this - m.

Returns: *this.

constexpr year_month_day& year_month_day::operator+=(const years& y) noexcept;

Effects: *this = *this + y.

Returns: *this.

constexpr year_month_day& year_month_day::operator-=(const years& y) noexcept;

Effects: *this = *this - y.

Returns: *this.

constexpr chrono::year year() const noexcept;

Returns: y_.

constexpr chrono::month month() const noexcept;

Returns: m_.

constexpr chrono::day day() const noexcept;

Returns: d_.

constexpr operator sys_days() const noexcept;

Returns: If ok(), returns a sys_days holding a count of days from the sys_days epoch to *this (a negative value if *this represents a date prior to the sys_days epoch). Otherwise, if y_.ok() && m_.ok() is true, returns sys_days{y_/m_/1d} + (d_ - 1d). Otherwise the value returned is unspecified.

Remarks: A sys_days in the range [days{-12687428}, days{11248737}] which is converted to a year_month_day has the same value when converted back to a sys_days.

[Example 1:]

static_assert(year_month_day{sys_days{2017y/January/0}} == 2016y/December/31);
static_assert(year_month_day{sys_days{2017y/January/31}} == 2017y/January/31);
static_assert(year_month_day{sys_days{2017y/January/32}} == 2017y/February/1);

—end example]

constexpr explicit operator local_days() const noexcept;

Returns: local_days{sys_days{*this}.time_since_epoch()}.

constexpr bool ok() const noexcept;

Returns: If y_.ok() is true, and m_.ok() is true, and d_ is in the range [1d, (y_/m_/last).day()], then returns true; otherwise returns false.
27.8.14.3  Non-member functions

castexpr bool operator==(const year_month_day& x, const year_month_day& y) noexcept;
1 Returns: x.year() == y.year() && x.month() == y.month() && x.day() == y.day().

castexpr strong_ordering operator<=>(const year_month_day& x, const year_month_day& y) noexcept;
2 Effects: Equivalent to:
   if (auto c = x.year() <=> y.year(); c != 0) return c;
   if (auto c = x.month() <=> y.month(); c != 0) return c;
   return x.day() <=> y.day();

constexpr year_month_day operator+(const year_month_day& ymd, const months& dm) noexcept;
3 Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).
4 Returns: (ymd.year() / ymd.month() + dm) / ymd.day().
5 [Note 1: If ymd.day() is in the range [1d, 28d], ok() will return true for the resultant year_month_day. —end note]
6
constexpr year_month_day operator+(const months& dm, const year_month_day& ymd) noexcept;
7 Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).
8 Returns: ymd + dm.

constexpr year_month_day operator-(const year_month_day& ymd, const months& dm) noexcept;
9 Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).
10 Returns: ymd + (-dm).
11
constexpr year_month_day operator+(const year_month_day& ymd, const years& dy) noexcept;
12 Returns: ymd + dy.
13
constexpr year_month_day operator+(const years& dy, const year_month_day& ymd) noexcept;
14
15 template<class charT, class traits>
   basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, const year_month_day& ymd);
16 Effects: Equivalent to:
   return os << (ymd.ok() ?
      format(STATICALLY-WIDEN<charT>("{:%F}"), ymd) :
      format(STATICALLY-WIDEN<charT>("{:%F} is not a valid date"), ymd));

template<class charT, class traits, class Alloc = allocator<charT>>
   basic_istream<charT, traits>&
   from_stream(basic_istream<charT, traits>& is, const charT* fmt, year_month_day& ymd, basic_string<charT, traits, Alloc>* abbrev = nullptr, minutes* offset = nullptr);
17 Effects: Attempts to parse the input stream is into the year_month_day ymd using the format flags given in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid year_month_day, is.setstate(ios_base::failbit) is called and ymd is not modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null.
Class year_month_day_last

Overview

namespace std::chrono {
    class year_month_day_last {
        chrono::year y_; // exposition only
        chrono::month_day_last mdl_; // exposition only
    public:
        constexpr year_month_day_last(const chrono::year& y,
                                    const chrono::month_day_last& mdl) noexcept;
        constexpr year_month_day_last& operator+=(const months& m) noexcept;
        constexpr year_month_day_last& operator-=(const months& m) noexcept;
        constexpr year_month_day_last& operator+=(const years& y) noexcept;
        constexpr year_month_day_last& operator-=(const years& y) noexcept;
        constexpr chrono::year year() const noexcept;
        constexpr chrono::month month() const noexcept;
        constexpr chrono::month_day_last month_day_last() const noexcept;
        constexpr chrono::day day() const noexcept;
        constexpr operator sys_days() const noexcept;
        constexpr explicit operator local_days() const noexcept;
        constexpr bool ok() const noexcept;
    };
}

1 year_month_day_last represents the last day of a specific year and month. year_month_day_last is a field-based time point with a resolution of days, except that it is restricted to pointing to the last day of a year and month.

[Note 1: year_month_day_last supports years- and months-oriented arithmetic, but not days-oriented arithmetic. For the latter, there is a conversion to sys_days, which efficiently supports days-oriented arithmetic. — end note]

year_month_day_last meets the Cpp17EqualityComparable (Table 25) and Cpp17LessThanComparable (Table 26) requirements.

year_month_day_last is a trivially copyable and standard-layout class type.

Member functions

constexpr year_month_day_last(const chrono::year& y,
                              const chrono::month_day_last& mdl) noexcept;

Effects: Initializes y_ with y and mdl_ with mdl.

constexpr year_month_day_last& operator+=(const months& m) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Effects: *this = *this + m.

Returns: *this.

constexpr year_month_day_last& operator-=(const months& m) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Effects: *this = *this - m.

Returns: *this.

constexpr year_month_day_last& operator+=(const years& y) noexcept;

Effects: *this = *this + y.

Returns: *this.
constexpr year_month_day_last& operator-=(const years& y) noexcept;

Effects: *this = *this - y.

Returns: *this.

cconstexpr chrono::year year() const noexcept;

Returns: y_.

cconstexpr chrono::month month() const noexcept;

Returns: mdl_.month().

cconstexpr chrono::month_day_last month_day_last() const noexcept;

Returns: mdl_.

cconstexpr chrono::day day() const noexcept;

Returns: If ok() is true, returns a day representing the last day of the (year, month) pair represented by *this. Otherwise, the returned value is unspecified.

[Note 1: This value might be computed on demand. —end note]

cconstexpr operator sys_days() const noexcept;

Returns: sys_days{year()/month()/day()}.

cconstexpr explicit operator local_days() const noexcept;

Returns: local_days{sys_days{*this}.time_since_epoch()}.

cconstexpr bool ok() const noexcept;

Returns: y_.ok() && mdl_.ok().

27.8.15.3 Non-member functions

constexpr bool operator==(const year_month_day_last& x, const year_month_day_last& y) noexcept;

Returns: x.year() == y.year() && x.month_day_last() == y.month_day_last().

cconstexpr strong_ordering operator<=>(const year_month_day_last& x, const year_month_day_last& y) noexcept;

Effects: Equivalent to:

if (auto c = x.year() <=> y.year(); c != 0) return c;

return x.month_day_last() <=> y.month_day_last();

cconstexpr year_month_day_last
operator+(const year_month_day_last& ymdl, const months& dm) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Returns: (ymdl.year() / ymdl.month() + dm) / last.

cconstexpr year_month_day_last
operator+(const months& dm, const year_month_day_last& ymdl) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Returns: ymdl + dm.

cconstexpr year_month_day_last
operator-(const year_month_day_last& ymdl, const months& dm) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Returns: ymdl + (-dm).
constexpr year_month_day_last
operator+(const year_month_day_last& ymdl, const years& dy) noexcept;  
\textit{Returns:}\ {ymdl.year() + dy, ymdl.month_day_last().}

constexpr year_month_day_last
operator+(const years& dy, const year_month_day_last& ymdl) noexcept;  
\textit{Returns:}\ ymdl + dy.

constexpr year_month_day_last
operator-(const year_month_day_last& ymdl, const years& dy) noexcept;  
\textit{Returns:}\ ymdl + (-dy).

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_day_last& ymdl);
\textit{Effects:}\ Equivalent to:
return os << format(os.getloc(), \texttt{STATICALLY-WIDEN<charT>("{}/{}")},
ymdl.year(), ymdl.month_day_last());

27.8.16 Class year_month_weekday
[time.cal.ymwd]

27.8.16.1 Overview
[time.cal.ymwd.overview]

namespace std::chrono {
    class year_month_weekday {
        chrono::year y_;   // exposition only
        chrono::month m_;   // exposition only
        chrono::weekday_indexed wdi_; // exposition only

        public:
        year_month_weekday() = default;
        constexpr year_month_weekday(const chrono::year& y, const chrono::month& m,
            const chrono::weekday_indexed& wdi) noexcept;
        constexpr year_month_weekday(const sys_days& dp) noexcept;
        constexpr explicit year_month_weekday(const local_days& dp) noexcept;
        constexpr year_month_weekday& operator+=(const months& m) noexcept;
        constexpr year_month_weekday& operator-=(const months& m) noexcept;
        constexpr year_month_weekday& operator+=(const years& y) noexcept;
        constexpr year_month_weekday& operator-=(const years& y) noexcept;
        constexpr chrono::year year() const noexcept;
        constexpr chrono::month month() const noexcept;
        constexpr chrono::weekday weekday() const noexcept;
        constexpr unsigned index() const noexcept;
        constexpr chrono::weekday_indexed weekday_indexed() const noexcept;
        constexpr operator sys_days() const noexcept;
        constexpr explicit operator local_days() const noexcept;
        constexpr bool ok() const noexcept;
    };
}
1 \textit{year_month_weekday} represents a specific year, month, and \(n^{th}\) weekday of the month. \textit{year_month_weekday} is a field-based time point with a resolution of days.

\[\text{Note 1:} \textit{year_month_weekday} \text{ supports } \textit{years-} \text{ and } \textit{months-} \text{ oriented arithmetic, but not } \textit{days-} \text{ oriented arithmetic. For the latter, there is a conversion to } \textit{sys_days}, \text{ which efficiently supports } \textit{days-} \text{ oriented arithmetic.} \]  
\[\text{— end note}\]

\textit{year_month_weekday} meets the \textit{Cpp17EqualityComparable} (Table 25) requirements.

2 \textit{year_month_weekday} is a trivially copyable and standard-layout class type.
27.8.16.2 Member functions

```cpp
constexpr year_month_weekday(const chrono::year& y, const chrono::month& m,
    const chrono::weekday_indexed& wdi) noexcept;

Effects: Initializes y_ with y, m_ with m, and wdi_ with wdi.
```

```cpp
constexpr year_month_weekday(const sys_days& dp) noexcept;
```

Effects: Constructs an object of type year_month_weekday which corresponds to the date represented by dp.

Remarks: For any value ymdl of type year_month_weekday for which ymdl.ok() is true, ymdl == year_month_weekday{sys_days{ymdl}} is true.

```cpp
constexpr explicit year_month_weekday(const local_days& dp) noexcept;
```

Effects: Equivalent to constructing with sys_days{dp.time_since_epoch()}.

```cpp
constexpr year_month_weekday& operator+=(const months& m) noexcept;
```

Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Effects: *this = *this + m.

Returns: *this.

```cpp
constexpr year_month_weekday& operator-=(const months& m) noexcept;
```

Constraints: If the argument supplied by the caller for the months parameter is convertible to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Effects: *this = *this - m.

Returns: *this.

```cpp
constexpr year_month_weekday& operator+=(const years& y) noexcept;
```

Effects: *this = *this + y.

Returns: *this.

```cpp
constexpr year_month_weekday& operator-=(const years& y) noexcept;
```

Effects: *this = *this - y.

Returns: *this.

```cpp
constexpr chrono::year year() const noexcept;
```

Returns: y_.

```cpp
constexpr chrono::month month() const noexcept;
```

Returns: m_.

```cpp
constexpr chrono::weekday weekday() const noexcept;
```

Returns: wdi_.weekday().

```cpp
constexpr unsigned index() const noexcept;
```

Returns: wdi_.index().

```cpp
constexpr chrono::weekday_indexed weekday_indexed() const noexcept;
```

Returns: wdi_.

```cpp
constexpr operator sys_days() const noexcept;
```

Returns: If y_.ok() && m_.ok() && wdi_.weekday().ok(), returns a sys_days that represents the date (index() - 1) * 7 days after the first weekday() of year()/month(). If index() is 0 the returned sys_days represents the date 7 days prior to the first weekday() of year()/month(). Otherwise the returned value is unspecified.
constexpr explicit operator local_days() const noexcept;

Returns: local_days{sys_days{*this}.time_since_epoch()}.

castexpr bool ok() const noexcept;

Returns: If any of y_.ok(), m_.ok(), or wdl_.ok() is false, returns false. Otherwise, if *this
represents a valid date, returns true. Otherwise, returns false.

27.8.16.3 Non-member functions

constexpr bool operator==(const year_month_weekday& x, const year_month_weekday& y) noexcept;

Returns:

x.year() == y.year() && x.month() == y.month() && x.weekday_indexed() == y.weekday_indexed()

constexpr year_month_weekday operator+(const year_month_weekday& ymwd, const months& dm) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible
to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Returns: (ymwd.year() / ymwd.month() + dm) / ymwd.weekday_indexed().

castexpr year_month_weekday operator+(const months& dm, const year_month_weekday& ymwd) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible
to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Returns: ymwd + dm.

castexpr year_month_weekday operator-(const year_month_weekday& ymwd, const months& dm) noexcept;

Constraints: If the argument supplied by the caller for the months parameter is convertible
to years, its implicit conversion sequence to years is worse than its implicit conversion sequence to months (12.4.4.3).

Returns: ymwd + (-dm).

castexpr year_month_weekday operator+(const year_month_weekday& ymwd, const years& dy) noexcept;

Returns: {ymwd.year()+dy, ymwd.month(), ymwd.weekday_indexed()}.

castexpr year_month_weekday operator+(const years& dy, const year_month_weekday& ymwd) noexcept;

Returns: ymwd + dy.

castexpr year_month_weekday operator-(const year_month_weekday& ymwd, const years& dy) noexcept;

Returns: ymwd + (-dy).

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_weekday& ymwd);

Effects: Equivalent to:

return os << format(os.getloc(), "statically-widen<charT>("{}\}/{}\}),

ymwd.year(), ymwd.month(), ymwd.weekday_indexed());

27.8.17 Class year_month_weekday_last

27.8.17.1 Overview

namespace std::chrono {

class year_month_weekday_last {

chrono::year y_; // exposition only
chrono::month m_; // exposition only
chrono::weekday_last wdl_; // exposition only

public:

castexpr year_month_weekday_last(const chrono::year& y, const chrono::month& m,
const chrono::weekday_last& wdl) noexcept;

§ 27.8.17.1
```cpp
constexpr year_month_weekday_last& operator+=(const months& m) noexcept;
constexpr year_month_weekday_last& operator-=(const months& m) noexcept;
constexpr year_month_weekday_last& operator+=(const years& y) noexcept;
constexpr year_month_weekday_last& operator-=(const years& y) noexcept;

constexpr chrono::year year() const noexcept;
constexpr chrono::month month() const noexcept;
constexpr chrono::weekday weekday() const noexcept;
constexpr chrono::weekday_last weekday_last() const noexcept;
constexpr operator sys_days() const noexcept;
constexpr explicit operator local_days() const noexcept;
constexpr bool ok() const noexcept;
}
```

1. `year_month_weekday_last` represents a specific year, month, and last weekday of the month. `year_month_weekday_last` is a field-based time point with a resolution of `days`, except that it is restricted to pointing to the last weekday of a year and month.

[Note 1: `year_month_weekday_last` supports `years`- and `months`-oriented arithmetic, but not `days`-oriented arithmetic. For the latter, there is a conversion to `sys_days`, which efficiently supports `days`-oriented arithmetic. — end note]

`year_month_weekday_last` meets the `Cpp17EqualityComparable` (Table 25) requirements.

2. `year_month_weekday_last` is a trivially copyable and standard-layout class type.

### 27.8.17.2 Member functions

`year_month_weekday_last` represents a specific year, month, and last weekday of the month. `year_month_weekday_last` is a field-based time point with a resolution of `days`, except that it is restricted to pointing to the last weekday of a year and month.

[Note 1: `year_month_weekday_last` supports `years`- and `months`-oriented arithmetic, but not `days`-oriented arithmetic. For the latter, there is a conversion to `sys_days`, which efficiently supports `days`-oriented arithmetic. — end note]

`year_month_weekday_last` meets the `Cpp17EqualityComparable` (Table 25) requirements.

2. `year_month_weekday_last` is a trivially copyable and standard-layout class type.

### 27.8.17.2 Member functions

```cpp
constexpr year_month_weekday_last(const chrono::year& y, const chrono::month& m, const chrono::weekday_last& wdl) noexcept;
    Effects: Initializes `y_` with `y`, `m_` with `m`, and `wdl_` with `wdl`.

constexpr year_month_weekday_last& operator+=(const months& m) noexcept;
    Constraints: If the argument supplied by the caller for the `months` parameter is convertible to `years`, its implicit conversion sequence to `years` is worse than its implicit conversion sequence to `months` (12.4.4.3).
    Effects: `*this = *this + m`.
    Returns: `*this`.

constexpr year_month_weekday_last& operator-=(const months& m) noexcept;
    Constraints: If the argument supplied by the caller for the `months` parameter is convertible to `years`, its implicit conversion sequence to `years` is worse than its implicit conversion sequence to `months` (12.4.4.3).
    Effects: `*this = *this - m`.
    Returns: `*this`.

constexpr year_month_weekday_last& operator+=(const years& y) noexcept;
    Effects: `*this = *this + y`.
    Returns: `*this`.

constexpr year_month_weekday_last& operator-=(const years& y) noexcept;
    Effects: `*this = *this - y`.
    Returns: `*this`.

constexpr chrono::year year() const noexcept;
    Returns: `y_`.

constexpr chrono::month month() const noexcept;
    Returns: `m_`.

constexpr chrono::weekday weekday() const noexcept;
    Returns: `wdl_.weekday()`.
```
constexpr chrono::weekday_last weekday_last() const noexcept;

Returns: wdl_.

constexpr operator sys_days() const noexcept;

Returns: If ok() == true, returns a sys_days that represents the last weekday() of year()/month(). Otherwise the returned value is unspecified.

constexpr explicit operator local_days() const noexcept;

Returns: local_days{sys_days{*this}.time_since_epoch()}.

constexpr bool ok() const noexcept;

Returns: y_.ok() && m_.ok() && wdl_.ok().
27.8.18 Conventional syntax operators

A set of overloaded `operator/` functions provides a conventional syntax for the creation of civil calendar dates.

[Note 1: The year, month, and day are accepted in any of the following 3 orders:

```
year/month/day
month/day/year
day/month/year
```

Anywhere a `day` is required, any of the following can also be specified:

```
last
weekday[i]
weekday[last]
```

—end note]

[Note 2: Partial-date types such as `year_month` and `month_day` can be created by not applying the second division operator for any of the three orders. For example:

```
year_month ym = 2015y/April;
month_day md1 = April/4;
month_day md2 = 4d/April;
```

—end note]

[Example 1:

```
auto a = 2015/4/4; // a == int(125)
auto b = 2015y/4/4; // b == year_month_day{year(2015), month(4), day(4)}
auto c = 2015y/4d/April; // error: no viable operator/ for first /
auto d = 2015/April/4; // error: no viable operator/ for first /
```

—end example]

```
constexpr year_month
operator/(const year& y, const month& m) noexcept;
```

Returns: `{y, m}`.

```
constexpr year_month
operator/(const year& y, int m) noexcept;
```

Returns: `y / month(m)`.

```
constexpr month_day
operator/(const month& m, const day& d) noexcept;
```

Returns: `{m, d}`.

```
constexpr month_day
operator/(const month& m, int d) noexcept;
```

Returns: `m / day(d)`.

```
constexpr month_day
operator/(int m, const day& d) noexcept;
```

Returns: `month(m) / d`.

```
constexpr month_day
operator/(const day& d, const month& m) noexcept;
```

Returns: `m / d`.

```
constexpr month_day
operator/(const day& d, int m) noexcept;
```

Returns: `month(m) / d`.

```
constexpr month_day_last
operator/(const month& m, last_spec) noexcept;
```

Returns: `month_day_last(m)`. 
constexpr month_day_last
operator/(int m, last_spec) noexcept;

Returns: month(m) / last.

constexpr month_day_last
operator/(last_spec, const month& m) noexcept;

Returns: m / last.

constexpr month_day_last
operator/(last_spec, int m) noexcept;

Returns: month(m) / last.

constexpr month_weekday
operator/(const month& m, const weekday_indexed& wdi) noexcept;

Returns: {m, wdi}.

constexpr month_weekday
operator/(int m, const weekday_indexed& wdi) noexcept;

Returns: month(m) / wdi.

constexpr month_weekday
operator/(const weekday_indexed& wdi, const month& m) noexcept;

Returns: m / wdi.

constexpr month_weekday
operator/(const weekday_indexed& wdi, int m) noexcept;

Returns: month(m) / wdi.

constexpr month_weekday_last
operator/(const month& m, const weekday_last& wdl) noexcept;

Returns: {m, wdl}.

constexpr month_weekday_last
operator/(int m, const weekday_last& wdl) noexcept;

Returns: month(m) / wdl.

constexpr month_weekday_last
operator/(const weekday_last& wdl, const month& m) noexcept;

Returns: m / wdl.

constexpr month_weekday_last
operator/(const weekday_last& wdl, int m) noexcept;

Returns: month(m) / wdl.

constexpr year_month_day
operator/(const year_month& ym, const day& d) noexcept;

Returns: {ym.year(), ym.month(), d}.

constexpr year_month_day
operator/(const year_month& ym, int d) noexcept;

Returns: ym / day(d).

constexpr year_month_day
operator/(const year_month& ym, const month_day& md) noexcept;

Returns: y / md.month() / md.day()..
constexpr year_month_day
operator/(const month_day& md, const year& y) noexcept;

Returns: y / md.

constexpr year_month_day
operator/(const month_day& md, int y) noexcept;

Returns: year(y) / md.

constexpr year_month_day_last
operator/(const year_month& ym, last_spec) noexcept;

Returns: {ym.year(), month_day_last{ym.month()}}.

constexpr year_month_day_last
operator/(const year& y, const month_day_last& mdl) noexcept;

Returns: {y, mdl}.

constexpr year_month_day_last
operator/(int y, const month_day_last& mdl) noexcept;

Returns: year(y) / mdl.

constexpr year_month_day_last
operator/(const month_day_last& mdl, const year& y) noexcept;

Returns: y / mdl.

constexpr year_month_day_last
operator/(const month_day_last& mdl, int y) noexcept;

Returns: year(y) / mdl.

constexpr year_month_weekday
operator/(const year_month& ym, const weekday_indexed& wdi) noexcept;

Returns: {ym.year(), ym.month(), wdi}.

constexpr year_month_weekday
operator/(const year& y, const month_weekday& mwd) noexcept;

Returns: {y, mwd.month(), mwd.weekday_indexed()}.

constexpr year_month_weekday
operator/(int y, const month_weekday& mwd) noexcept;

Returns: year(y) / mwd.

constexpr year_month_weekday
operator/(const month_weekday& mwd, const year& y) noexcept;

Returns: y / mwd.

constexpr year_month_weekday
operator/(const month_weekday& mwd, int y) noexcept;

Returns: year(y) / mwd.

constexpr year_month_weekday_last
operator/(const year_month& ym, const weekday_last& wdl) noexcept;

Returns: {ym.year(), ym.month(), wdl}.

constexpr year_month_weekday_last
operator/(const year& y, const month_weekday_last& mwdl) noexcept;

Returns: {y, mwdl.month(), mwdl.weekday_last()}.

constexpr year_month_weekday_last
operator/(int y, const month_weekday_last& mwdl) noexcept;

Returns: year(y) / mwdl.
constexpr year_month_weekday_last
operator/(const month_weekday_last& mwdl, const year& y) noexcept;

Returns: \(y / mwdl\).

cconstexpr year_month_weekday_last
operator/(const month_weekday_last& mwdl, int y) noexcept;

Returns: \(year(y) / mwdl\).

27.9 Class template hh_mm_ss [time.hms]

27.9.1 Overview [time.hms.overview]

namespace std::chrono {
    template<class Duration> class hh_mm_ss {
        public:
            static constexpr unsigned fractional_width = see below;
            using precision = see below;

            constexpr hh_mm_ss() noexcept : hh_mm_ss(Duration::zero()) {} 
            constexpr explicit hh_mm_ss(Duration d);

            constexpr bool is_negative() const noexcept;
            constexpr chrono::hours hours() const noexcept;
            constexpr chrono::minutes minutes() const noexcept;
            constexpr chrono::seconds seconds() const noexcept;
            constexpr precision subseconds() const noexcept;
            constexpr explicit operator precision() const noexcept;
            constexpr precision to_duration() const noexcept;

        private:
            bool is_neg;  // exposition only
            chrono::hours h;  // exposition only
            chrono::minutes m;  // exposition only
            chrono::seconds s;  // exposition only
            precision ss;  // exposition only
    };
}

1 The hh_mm_ss class template splits a duration into a multi-field time structure _hours:minutes:seconds_ and possibly subseconds, where subseconds will be a duration unit based on a non-positive power of 10. The Duration template parameter dictates the precision to which the time is split. A hh_mm_ss models negative durations with a distinct is_negative getter that returns true when the input duration is negative. The individual duration fields always return non-negative durations even when is_negative() indicates the structure is representing a negative duration.

2 If Duration is not an instance of duration, the program is ill-formed.

27.9.2 Members [time.hms.members]

static constexpr unsigned fractional_width = see below;

1 fractional_width is the number of fractional decimal digits represented by precision. fractional_width has the value of the smallest possible integer in the range [0, 18] such that precision will exactly represent all values of Duration. If no such value of fractional_width exists, then fractional_width is 6.

[Example 1: See Table 98 for some durations, the resulting fractional_width, and the formatted fractional second output of Duration{1}.]

<table>
<thead>
<tr>
<th>Duration</th>
<th>fractional_width</th>
<th>Formatted fractional second output</th>
</tr>
</thead>
<tbody>
<tr>
<td>hours, minutes, and seconds</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>milliseconds</td>
<td>3</td>
<td>0.001</td>
</tr>
<tr>
<td>microseconds</td>
<td>6</td>
<td>0.000001</td>
</tr>
</tbody>
</table>

§ 27.9.2
Table 98: Examples for fractional_width (continued)

<table>
<thead>
<tr>
<th>Duration</th>
<th>fractional_width</th>
<th>Formatted fractional second output</th>
</tr>
</thead>
<tbody>
<tr>
<td>nanoseconds</td>
<td>9</td>
<td>0.000000001</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 2&gt;&gt;</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 3&gt;&gt;</td>
<td>6</td>
<td>0.333333</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 4&gt;&gt;</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 5&gt;&gt;</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 6&gt;&gt;</td>
<td>6</td>
<td>0.166666</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 7&gt;&gt;</td>
<td>6</td>
<td>0.142857</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 8&gt;&gt;</td>
<td>3</td>
<td>0.125</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 9&gt;&gt;</td>
<td>6</td>
<td>0.111111</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 10&gt;&gt;</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;756, 625&gt;&gt;</td>
<td>4</td>
<td>0.2096</td>
</tr>
</tbody>
</table>

—end example—

using precision = see below;

2

precision is

duration<common_type_t<Duration::rep, seconds::rep>, ratio<1, 10>>

constexpr explicit hh_mm_ss(Duration d);

3

Effects: Constructs an object of type hh_mm_ss which represents the Duration d with precision precision.

1.1 — Initializes is_neg with d < Duration::zero().
1.2 — Initializes h with duration_cast<chrono::hours>(abs(d)).
1.3 — Initializes m with duration_cast<chrono::minutes>(abs(d) - hours()).
1.4 — Initializes s with duration_cast<chrono::seconds>(abs(d) - hours() - minutes()).
1.5 — If treat_as_floating_point_v<precision::rep> is true, initializes ss with abs(d) - hours() - minutes() - seconds(). Otherwise, initializes ss with duration_cast<precision>(abs(d) - hours() - minutes() - seconds()).

[Note 1: When precision::rep is integral and precision::period is ratio<1>, subseconds() always returns a value equal to 0s. — end note]

4

Postconditions: If treat_as_floating_point_v<precision::rep> is true, to_duration() returns d, otherwise to_duration() returns duration_cast<precision>(d).

constexpr bool is_negative() const noexcept;

Returns: is_neg.

constexpr chrono::hours hours() const noexcept;

Returns: h.

constexpr chrono::minutes minutes() const noexcept;

Returns: m.

constexpr chrono::seconds seconds() const noexcept;

Returns: s.

constexpr precision subseconds() const noexcept;

Returns: ss.

constexpr precision to_duration() const noexcept;

Returns: If is_neg, returns -(h + m + s + ss), otherwise returns h + m + s + ss.

constexpr explicit operator precision() const noexcept;

Returns: to_duration().

§ 27.9.2
27.9.3 Non-members

```cpp
template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const hh_mm_ss<Duration>& hms);
```

Effects: Equivalent to:

```cpp
return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{:T}"), hms);
```

[Example 1:](#)
```
for (auto ms : {-4083007ms, 4083007ms, 65745123ms}) {
    hh_mm_ss hms{ms};
    cout << hms << '\n';
}
cout << hh_mm_ss{65745s} << '\n';
```

Produces the output (assuming the 'C' locale):

```
-01:08:03.007
01:08:03.007
18:15:45.123
18:15:45
```

— end example —

27.10 12/24 hours functions

These functions aid in translating between a 12h format time of day and a 24h format time of day.

```cpp
constexpr bool is_am(const hours& h) noexcept;
```

Returns: 0h <= h && h <= 11h.

```cpp
constexpr bool is_pm(const hours& h) noexcept;
```

Returns: 12h <= h && h <= 23h.

```cpp
constexpr hours make12(const hours& h) noexcept;
```

Returns: The 12-hour equivalent of h in the range [1h, 12h]. If h is not in the range [0h, 23h], the value returned is unspecified.

```cpp
constexpr hours make24(const hours& h, bool is_pm) noexcept;
```

Returns: If is_pm is false, returns the 24-hour equivalent of h in the range [0h, 11h], assuming h represents an ante meridiem hour. Otherwise, returns the 24-hour equivalent of h in the range [12h, 23h], assuming h represents a post meridiem hour. If h is not in the range [1h, 12h], the value returned is unspecified.

27.11 Time zones

27.11.1 In general

27.11 describes an interface for accessing the IANA Time Zone Database that interoperates with sys_time and local_time. This interface provides time zone support to both the civil calendar types (27.8) and to user-defined calendars.

27.11.2 Time zone database

27.11.2.1 Class tzdb

```cpp
namespace std::chrono {
    struct tzdb {
        string version;
        vector<time_zone> zones;
        vector<time_zone_link> links;
        vector<leap_second> leap_seconds;

        const time_zone* locate_zone(string_view tz_name) const;
        const time_zone* current_zone() const;
    };
}
```
Each vector in a tzdb object is sorted to enable fast lookup.

```
const time_zone* locate_zone(string_view tz_name) const;
```

**Returns:**

1. If zones contains an element tz for which tz.name() == tz_name, a pointer to tz;
2. otherwise, if links contains an element tz_l for which tz_l.name() == tz_name, then a pointer to the element tz of zones for which tz.name() == tz_l.target().

\[\text{Note 1: A time_zone_link specifies an alternative name for a time_zone. \textit{end note}}\]

**Throws:** If a const time_zone* cannot be found as described in the **Returns** element, throws a runtime_error.

\[\text{Note 2: On non-exceptional return, the return value is always a pointer to a valid time_zone. \textit{end note}}\]

```
const time_zone* current_zone() const;
```

**Returns:** A pointer to the time zone which the computer has set as its local time zone.

### 27.11.2.2 Class tzdb_list

```
namespace std::chrono {
    class tzdb_list {
    public:
        tzdb_list(const tzdb_list&) = delete;
        tzdb_list& operator=(const tzdb_list&) = delete;

        // unspecified additional constructors
        class const_iterator;
        const tzdb& front() const noexcept;
        const_iterator erase_after(const_iterator p);
        const_iterator begin() const noexcept;
        const_iterator end() const noexcept;
        const_iterator cbegin() const noexcept;
        const_iterator cend() const noexcept;
    }
}
```

The tzdb_list database is a singleton; the unique object of type tzdb_list can be accessed via the get_tzdb_list() function.

\[\text{Note 1: This access is only needed for those applications that need to have long uptimes and have a need to update the time zone database while running. Other applications can implicitly access the front() of this list via the read-only namespace scope functions get_tzdb(), locate_zone(), and current_zone(). \textit{end note}}\]

The tzdb_list object contains a list of tzdb objects.

```
tzdb_list::const_iterator is a constant iterator which meets the Cpp17ForwardIterator requirements and has a value type of tzdb.
```

```
const tzdb& front() const noexcept;
```

**Synchronization:** This operation is thread-safe with respect to reload_tzdb().

\[\text{Note 2: reload_tzdb() pushes a new tzdb onto the front of this container. \textit{end note}}\]

**Returns:** A reference to the first tzdb in the container.

```
const_iterator erase_after(const_iterator p);
```

**Preconditions:** The iterator following p is dereferenceable.

**Effects:** Erases the tzdb referred to by the iterator following p.

**Postconditions:** No pointers, references, or iterators are invalidated except those referring to the erased tzdb.
[Note 3: It is not possible to erase the tzdb referred to by begin(). — end note]

8 Return: An iterator pointing to the element following the one that was erased, or end() if no such element exists.

9 Throws: Nothing.

const_iterator begin() const noexcept;

Returns: An iterator referring to the first tzdb in the container.

const_iterator end() const noexcept;

Returns: An iterator referring to the position one past the last tzdb in the container.

const_iterator cbegin() const noexcept;

Returns: begin().

const_iterator cend() const noexcept;

Returns: end().

27.11.2.3 Time zone database access [time.zone.db.access]


tzdb_list& get_tzdb_list();

1 Effects: If this is the first access to the time zone database, initializes the database. If this call initializes
the database, the resulting database will be a tzdb_list holding a single initialized tzdb.

2 Synchronization: It is safe to call this function from multiple threads at one time.

3 Returns: A reference to the database.

4 Throws: runtime_error if for any reason a reference cannot be returned to a valid tzdb_list containing
one or more valid tzdb.

const tzdb& get_tzdb();

5 Returns: get_tzdb_list().front().

const time_zone* locate_zone(string_view tz_name);  

6 Returns: get_tzdb().locate_zone(tz_name).

7 [Note 1: The time zone database will be initialized if this is the first reference to the database. — end note]

const time_zone* current_zone();

8 Returns: get_tzdb().current_zone().

27.11.2.4 Remote time zone database support [time.zone.db.remote]

The local time zone database is that supplied by the implementation when the program first accesses
the database, for example via current_zone(). While the program is running, the implementation may choose
to update the time zone database. This update shall not impact the program in any way unless the program
calls the functions in this subclause. This potentially updated time zone database is referred to as the remote
time zone database.

const tzdb& reload_tzdb();

2 Effects: This function first checks the version of the remote time zone database. If the versions of the
local and remote databases are the same, there are no effects. Otherwise the remote database is pushed
to the front of the tzdb_list accessed by get_tzdb_list().

3 Synchronization: This function is thread-safe with respect to get_tzdb_list().front() and get_tzdb_list().erase_after().

4 Postconditions: No pointers, references, or iterators are invalidated.

5 Returns: get_tzdb_list().front().

6 Throws: runtime_error if for any reason a reference cannot be returned to a valid tzdb.
string remote_version();

Returns: The latest remote database version.

[Note 1: This can be compared with get_tzdb().version to discover if the local and remote databases are equivalent. — end note]

27.11.3 Exception classes

27.11.3.1 Class nonexistent_local_time

namespace std::chrono {
    class nonexistent_local_time : public runtime_error {
        public:
            template<class Duration>
                nonexistent_local_time(const local_time<Duration>& tp, const local_info& i);
        }
    }
	nonexistent_local_time is thrown when an attempt is made to convert a non-existent local_time to a
sys_time without specifying choose::earliest or choose::latest.

template<class Duration>
    nonexistent_local_time(const local_time<Duration>& tp, const local_info& i);

Preconditions: i.result == local_info::nonexistent is true.

Effects: Initializes the base class with a sequence of char equivalent to that produced by os.str() initialized as shown below:

    os << tp << " is in a gap between\n"
    << local_seconds{i.first.end.time_since_epoch()} + i.first.offset << ', '
    << i.first.abbrev << " and\n"
    << local_seconds{i.second.begin.time_since_epoch()} + i.second.offset << ', '
    << i.second.abbrev
    << " which are both equivalent to\n"
    << i.first.end << " UTC";

[Example 1:]
#include <chrono>
#include <iostream>

int main() {
    using namespace std::chrono;
    try {
        auto zt = zoned_time("America/New_York",
            local_days{Sunday[2]/March/2016} + 2h + 30min);
    } catch (const nonexistent_local_time& e) {
        std::cout << e.what() << '\n';
    }
}

Produces the output:

2016-03-13 02:30:00 is in a gap between
2016-03-13 02:00:00 EST and
2016-03-13 03:00:00 EDT which are both equivalent to
2016-03-13 07:00:00 UTC

— end example]

27.11.3.2 Class ambiguous_local_time

namespace std::chrono {
    class ambiguous_local_time : public runtime_error {
        public:
            template<class Duration>
                ambiguous_local_time(const local_time<Duration>& tp, const local_info& i);
        }
    }

§ 27.11.3.2
ambiguous_local_time is thrown when an attempt is made to convert an ambiguous local_time to a sys_time without specifying choose::earliest or choose::latest.

template<class Duration>
ambiguous_local_time(const local_time<Duration>& tp, const local_info& i);

Preconditions: i.result == local_info::ambiguous is true.

Effects: Initializes the base class with a sequence of char equivalent to that produced by os.str() initialized as shown below:

```cpp
ostringstream os;
os << tp << " is ambiguous. It could be\n" << tp << ' ' << i.first.abbrev << " == " << tp - i.first.offset << " UTC or\n" << tp << ' ' << i.second.abbrev << " == " << tp - i.second.offset << " UTC";
```

```cpp
#include <chrono>
#include <iostream>

int main() {
    using namespace std::chrono;
    try {
        auto zt = zoned_time{"America/New_York",
                             local_days{Sunday[1]/November/2016} + 1h + 30min};
    } catch (const ambiguous_local_time& e) {
        std::cout << e.what() << '\n';
    }
}
```

Produces the output:

```
2016-11-06 01:30:00 is ambiguous. It could be
2016-11-06 01:30:00 EDT == 2016-11-06 05:30:00 UTC or
2016-11-06 01:30:00 EST == 2016-11-06 06:30:00 UTC
```

—end example]

### 27.11.4 Information classes

#### 27.11.4.1 Class sys_info

A sys_info object can be obtained from the combination of a time_zone and either a sys_time or local_time. It can also be obtained from a zoned_time, which is effectively a pair of a time_zone and sys_time.

[Note 1: This type provides a low-level interface to time zone information. Typical conversions from sys_time to local_time will use this class implicitly, not explicitly. — end note]

The begin and end data members indicate that, for the associated time_zone and time_point, the offset and abbrev are in effect in the range (begin, end). This information can be used to efficiently iterate the transitions of a time_zone.

The offset data member indicates the UTC offset in effect for the associated time_zone and time_point. The relationship between local_time and sys_time is:

```
offset = local_time - sys_time
```

The save data member is extra information not normally needed for conversion between local_time and sys_time. If save != 0min, this sys_info is said to be on “daylight saving” time, and offset - save
suggests what offset this time_zone might use if it were off daylight saving time. However, this information should not be taken as authoritative. The only sure way to get such information is to query the time_zone with a time_point that returns a sys_info where save == 0min. There is no guarantee what time_point might return such a sys_info except that it is guaranteed not to be in the range [begin, end) (if save != 0min for this sys_info).

The abbrev data member indicates the current abbreviation used for the associated time_zone and time_point. Abbreviations are not unique among the time_zones, and so one cannot reliably map abbreviations back to a time_zone and UTC offset.

\[ \text{template<class charT, class traits>} \]
\[ \text{basic_ostream<charT, traits>&} \]
\[ \text{operator<<(basic_ostream<charT, traits>& os, const sys_info& r);} \]

7 Effects: Streams out the sys_info object r in an unspecified format.

8 Returns: os.

27.11.4.2 Class local_info

namespace std::chrono {
    struct local_info {
        static constexpr int unique = 0;
        static constexpr int nonexistent = 1;
        static constexpr int ambiguous = 2;

        int result;
        sys_info first;
        sys_info second;
    };
}

\[ \text{Note 1: This type provides a low-level interface to time zone information. Typical conversions from local_time to sys_time will use this class implicitly, not explicitly. — end note} \]

2 Describes the result of converting a local_time to a sys_time as follows:

\[ \text{(2.1)} \]
\[ \text{— When a local_time to sys_time conversion is unique, result == unique, first will be filled out with the correct sys_info, and second will be zero-initialized.} \]

\[ \text{(2.2)} \]
\[ \text{— If the conversion stems from a nonexistent local_time then result == nonexistent, first will be filled out with the sys_info that ends just prior to the local_time, and second will be filled out with the sys_info that begins just after the local_time.} \]

\[ \text{(2.3)} \]
\[ \text{— If the conversion stems from an ambiguous local_time, then result == ambiguous, first will be filled out with the sys_info that ends just after the local_time, and second will be filled out with the sys_info that starts just before the local_time.} \]

\[ \text{template<class charT, class traits>} \]
\[ \text{basic_ostream<charT, traits>&} \]
\[ \text{operator<<(basic_ostream<charT, traits>& os, const local_info& r);} \]

3 Effects: Streams out the local_info object r in an unspecified format.

4 Returns: os.

27.11.5 Class time_zone

27.11.5.1 Overview

namespace std::chrono {
    class time_zone {
    public:
        time_zone(time_zone&&) = default;
        time_zone& operator=(time_zone&&) = default;

        // unspecified additional constructors

        string_view name() const noexcept;
    };
}
template<class Duration> sys_info get_info(const sys_time<Duration>& st) const;
template<class Duration> local_info get_info(const local_time<Duration>& tp) const;

template<class Duration>
  sys_time<common_type_t<Duration, seconds>>
  to_sys(const local_time<Duration>& tp) const;

template<class Duration>
  sys_time<common_type_t<Duration, seconds>>
  to_sys(const local_time<Duration>& tp, choose z) const;

template<class Duration>
  local_time<common_type_t<Duration, seconds>>
  to_local(const sys_time<Duration>& tp) const;

A `time_zone` represents all time zone transitions for a specific geographic area. `time_zone` construction is unspecified, and performed as part of database initialization.

[Note 1: `const time_zone` objects can be accessed via functions such as `locate_zone`. —end note]

27.11.5.2 Member functions

`string_view name() const noexcept;`

1 Returns: The name of the `time_zone`.

2 [Example 1: "America/New_York". —end example]

`template<class Duration>
  sys_info get_info(const sys_time<Duration>& st) const;`

3 Returns: A `sys_info` `i` for which `st` is in the range `[i.begin, i.end)`. `template<class Duration>
  local_info get_info(const local_time<Duration>& tp) const;` 4 Returns: A `local_info` for `tp`

`template<class Duration>
  sys_time<common_type_t<Duration, seconds>>
  to_sys(const local_time<Duration>& tp) const;`

5 Returns: A `sys_time` that is at least as fine as `seconds`, and will be finer if the argument `tp` has finer precision. This `sys_time` is the UTC equivalent of `tp` according to the rules of this `time_zone`. `template<class Duration>
  sys_time<common_type_t<Duration, seconds>>
  to_sys(const local_time<Duration>& tp, choose z) const;`

6 Returns: A `sys_time` that is at least as fine as `seconds`, and will be finer if the argument `tp` has finer precision. This `sys_time` is the UTC equivalent of `tp` according to the rules of this `time_zone`. If the conversion from `tp` to a `sys_time` is ambiguous, returns the earlier `sys_time` if `z == choose::earliest`, and returns the later `sys_time` if `z == choose::latest`. If the `tp` represents a non-existent time between two UTC `time_points`, then the two UTC `time_points` will be the same, and that UTC `time_point` will be returned.

`template<class Duration>
  local_time<common_type_t<Duration, seconds>>
  to_local(const sys_time<Duration>& tp) const;`

7 Returns: The `local_time` associated with `tp` and this `time_zone`.
27.11.5.3 Non-member functions

```
bool operator==(const time_zone& x, const time_zone& y) noexcept;
```

1

*Returns:* `x.name() == y.name()`.

```
strong_ordering operator<=>(const time_zone& x, const time_zone& y) noexcept;
```

2

*Returns:* `x.name() <=> y.name()`.

27.11.6 Class template `zoned_traits`

```
namespace std::chrono {
  template<class T> struct zoned_traits {
  };
}
```

1

`zoned_traits` provides a means for customizing the behavior of `zoned_time<Duration, TimeZonePtr>` for the `zoned_time` default constructor, and constructors taking `string_view`. A specialization for `const time_zone*` is provided by the implementation:

```
namespace std::chrono {
  template<> struct zoned_traits<const time_zone*> {
    static const time_zone* default_zone();
    static const time_zone* locate_zone(string_view name);
  };
}
```

2

*Returns:* `std::chrono::locate_zone("UTC")`.

```
static const time_zone* default_zone();
```

2

*Returns:* `std::chrono::locate_zone(name)`.

27.11.7 Class template `zoned_time`

27.11.7.1 Overview

```
namespace std::chrono {
  template<class Duration, class TimeZonePtr = const time_zone*>
  class zoned_time {
    public:
      using duration = common_type_t<Duration, seconds>;

    private:
      TimeZonePtr zone_;                // exposition only
      sys_time<duration> tp_;           // exposition only

      using traits = zoned_traits<TimeZonePtr>; // exposition only

    public:
      zoned_time();
      zoned_time(const zoned_time&) = default;
      zoned_time& operator=(const zoned_time&) = default;
      
      zoned_time(const sys_time<Duration>& st);
      explicit zoned_time(TimeZonePtr z);
      explicit zoned_time(string_view name);

      template<class Duration2>
      zoned_time(const zoned_time<Duration2, TimeZonePtr>& zt);
      zoned_time(TimeZonePtr z, const sys_time<Duration>& st);
      zoned_time(string_view name, const sys_time<Duration>& st);

      zoned_time(TimeZonePtr z, const local_time<Duration>& tp);
      zoned_time(string_view name, const local_time<Duration>& tp);
      zoned_time(TimeZonePtr z, const local_time<Duration>& tp, choose c);
      zoned_time(string_view name, const local_time<Duration>& tp, choose c);
  }
}
```

§ 27.11.7.1
template<class Duration2, class TimeZonePtr2>
zM-time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& zt);

zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& zt, choose);

zoned_time(string_view name, const zoned_time<Duration2, TimeZonePtr2>& zt);

zoned_time(string_view name, const zoned_time<Duration2, TimeZonePtr2>& zt, choose);

zoned_time& operator=(const sys_time<Duration>& st);

zoned_time& operator=(const local_time<Duration>& ut);

operator sys_time<duration>() const;

explicit operator local_time<duration>() const;

TimeZonePtr get_time_zone() const;

local_time<duration> get_local_time() const;

sys_time<duration> get_sys_time() const;

sys_info get_info() const;

}

zoned_time() -> zoned_time<seconds>;

template<class Duration>
zed_time(sys_time<Duration>)
  -> zoned_time<common_type_t<Duration, seconds>>;

template<class TimeZonePtrOrName>
using time-zone-representation =
  // exposition only
  conditional_t<is_convertible_v<TimeZonePtrOrName, string_view>,
  const time_zone*,
  remove_cvref_t<TimeZonePtrOrName>>;

template<class TimeZonePtrOrName>
zed_time(TimeZonePtrOrName&&)
  -> zoned_time<seconds, time-zone-representation<TimeZonePtrOrName>>;

template<class TimeZonePtrOrName, class Duration>
zed_time(TimeZonePtrOrName&&, sys_time<Duration>)
  -> zoned_time<common_type_t<Duration, seconds>,
  time-zone-representation<TimeZonePtrOrName>>;

template<class TimeZonePtrOrName, class Duration>
zed_time(TimeZonePtrOrName&&, local_time<Duration>,
  choose = choose::earliest)
  -> zoned_time<common_type_t<Duration, seconds>,
  time-zone-representation<TimeZonePtrOrName>>;

template<class Duration, class TimeZonePtrOrName, class TimeZonePtr2>
zed_time(TimeZonePtrOrName&&, zoned_time<Duration, TimeZonePtr2>,
  choose = choose::earliest)
  -> zoned_time<common_type_t<Duration, seconds>,
  time-zone-representation<TimeZonePtrOrName>>;

1 zoned_time represents a logical pairing of a time_zone and a time_point with precision Duration. zoned_time maintains the invariant that it always refers to a valid time zone and represents a point in time that exists and is not ambiguous in that time zone.

2 If Duration is not a specialization of std::chrono::duration, the program is ill-formed.

3 Every constructor of zoned_time that accepts a string_view as its first parameter does not participate in class template argument deduction (12.4.2.9).
27.11.7.2 Constructors

zoned_time();
   
   Constraints: traits::default_zone() is a well-formed expression.
   
   Effects: Initializes zone_ with traits::default_zone() and default constructs tp_.

zoned_time(const sys_time<Duration>& st);
   
   Constraints: traits::default_zone() is a well-formed expression.
   
   Effects: Initializes zone_ with traits::default_zone() and tp_ with st.

explicit zoned_time(TimeZonePtr z);
   
   Preconditions: z refers to a time zone.
   
   Effects: Initializes zone_ with std::move(z) and default constructs tp_.

explicit zoned_time(string_view name);
   
   Constraints: traits::locate_zone(string_view{}) is a well-formed expression and zoned_time is constructible from the return type of traits::locate_zone(string_view{}).
   
   Effects: Initializes zone_ with traits::locate_zone(name) and default constructs tp_.

template<class Duration2>
   zoned_time(const zoned_time<Duration2, TimeZonePtr>& y);
   
   Constraints: is_convertible_v<sys_time<Duration2>, sys_time<Duration>> is true.
   
   Effects: Initializes zone_ with y.zone_ and tp_ with y.tp_.

zoned_time(TimeZonePtr z, const sys_time<Duration>& st);
   
   Preconditions: z refers to a time zone.
   
   Effects: Initializes zone_ with std::move(z) and tp_ with st.

zoned_time(string_view name, const sys_time<Duration>& st);
   
   Constraints: zoned_time is constructible from the return type of traits::locate_zone(name) and st.
   
   Effects: Equivalent to construction with {traits::locate_zone(name), st}.

zoned_time(TimeZonePtr z, const local_time<Duration>& tp);
   
   Constraints:
   
   is_convertible_v<
       decltype(declval<TimeZonePtr&>()->to_sys(local_time<Duration>{})) -> to_sys(local_time<Duration>{})),
       sys_time<duration>>
   
   is true.

   Preconditions: z refers to a time zone.
   
   Effects: Initializes zone_ with std::move(z) and tp_ with zone_->to_sys(tp).

zoned_time(string_view name, const local_time<Duration>& tp);
   
   Constraints: zoned_time is constructible from the return type of traits::locate_zone(name) and tp.
   
   Effects: Equivalent to construction with {traits::locate_zone(name), tp}.

zoned_time(TimeZonePtr z, const local_time<Duration>& tp, choose c);
   
   Constraints:
   
   is_convertible_v<
       decltype(declval<TimeZonePtr&>() -> to_sys(local_time<Duration>{}, choose::earliest)),
       sys_time<duration>>
   
   is true.

   Preconditions: z refers to a time zone.
Effects: Initializes zone_ with std::move(z) and tp_ with zone_->to_sys(tp, c).

zoned_time(string_view name, const local_time<Duration>& tp, choose c);

Constraints: zoned_time is constructible from the return type of traits::locate_zone(name), local_time<Duration>, and choose.

Effects: Equivalent to construction with {traits::locate_zone(name), tp, c}.

template<class Duration2, class TimeZonePtr2>
zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& y);

Constraints: is_convertible_v<sys_time<Duration2>, sys_time<Duration>> is true.

Preconditions: z refers to a valid time zone.

Effects: Equivalent to construction with {z, y}.

template<class Duration2, class TimeZonePtr2>
zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& y, choose);

Constraints: is_convertible_v<sys_time<Duration2>, sys_time<Duration>> is true.

Preconditions: z refers to a valid time zone.

Effects: Equivalent to construction with {z, y, c}.

[Note 1: The choose parameter has no effect. — end note]

template<class Duration2, class TimeZonePtr2>
zoned_time(string_view name, const zoned_time<Duration2, TimeZonePtr2>& y);

Constraints: zoned_time is constructible from the return type of traits::locate_zone(name) and the type zoned_time<Duration2, TimeZonePtr2>.

Effects: Equivalent to construction with {traits::locate_zone(name), y}.

template<class Duration2, class TimeZonePtr2>
zoned_time(string_view name, const zoned_time<Duration2, TimeZonePtr2>& y, choose c);

Constraints: zoned_time is constructible from the return type of traits::locate_zone(name), the type zoned_time<Duration2, TimeZonePtr2>, and the type choose.

Effects: Equivalent to construction with {traits::locate_zone(name), y, c}.

[Note 2: The choose parameter has no effect. — end note]

27.11.7.3 Member functions

zoned_time& operator=(const sys_time<Duration>& st);

Effects: After assignment, get_sys_time() == st. This assignment has no effect on the return value of get_time_zone().

Returns: *this.

zoned_time& operator=(const local_time<Duration>& lt);

Effects: After assignment, get_local_time() == lt. This assignment has no effect on the return value of get_time_zone().

Returns: *this.

operator sys_time<duration>() const;

Returns: get_sys_time().

explicit operator local_time<duration>() const;

Returns: get_local_time().

TimeZonePtr get_time_zone() const;

Returns: zone_.

local_time<duration> get_local_time() const;

Returns: zone_->to_local(tp_).
sys_time<duration> get_sys_time() const;

Returns: tp_.

sys_info get_info() const;

Returns: zone_->get_info(tp_).

27.11.7.4 Non-member functions

template<class Duration1, class Duration2, class TimeZonePtr>
bool operator==(const zoned_time<Duration1, TimeZonePtr>& x,
const zoned_time<Duration2, TimeZonePtr>& y);

Returns: x.zone_ == y.zone_ && x.tp_ == y.tp_.

template<class charT, class traits, class Duration, class TimeZonePtr>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
const zoned_time<Duration, TimeZonePtr>& t);

Effects: Streams the value returned from t.get_local_time() to os using the format "%F %T %Z".

Returns: os.

27.11.8 Class leap_second

27.11.8.1 Overview

namespace std::chrono {

class leap_second {

class leap_second(const leap_second&) = default;

leap_second& operator=(const leap_second&) = default;

// unspecified additional constructors

constexpr sys_seconds date() const noexcept;
constexpr seconds value() const noexcept;

};
}

Objects of type leap_second representing the date and value of the leap second insertions are constructed and stored in the time zone database when initialized.

[Example 1:

for (auto& l : get_tzdb().leap_seconds)
    if (l <= 2018y/March/17d)
        cout << l.date() << ": " << l.value() << \\
Produces the output:

1972-07-01 00:00:00: 1s
1973-01-01 00:00:00: 1s
1974-01-01 00:00:00: 1s
1975-01-01 00:00:00: 1s
1976-01-01 00:00:00: 1s
1977-01-01 00:00:00: 1s
1978-01-01 00:00:00: 1s
1979-01-01 00:00:00: 1s
1980-01-01 00:00:00: 1s
1981-07-01 00:00:00: 1s
1982-07-01 00:00:00: 1s
1983-07-01 00:00:00: 1s
1985-07-01 00:00:00: 1s
1988-01-01 00:00:00: 1s
1990-01-01 00:00:00: 1s
1991-01-01 00:00:00: 1s
1992-07-01 00:00:00: 1s
1993-07-01 00:00:00: 1s

§ 27.11.8.1 1324
27.11.8.2 Member functions

```
constexpr sys_seconds date() const noexcept;
```

*Returns:* The date and time at which the leap second was inserted.

```constexpr seconds value() const noexcept;
```

*Returns:* +1s to indicate a positive leap second or -1s to indicate a negative leap second.

*[Note 1: All leap seconds inserted up through 2019 were positive leap seconds. — end note]*

27.11.8.3 Non-member functions

```constexpr bool operator==(const leap_second& x, const leap_second& y) noexcept;
```

*Returns:* `x.date() == y.date()`.

```constexpr strong_ordering operator<=>(const leap_second& x, const leap_second& y) noexcept;
```

*Returns:* `x.date() <=> y.date()`.

```template<class Duration>
constexpr bool operator==(const leap_second& x, const sys_time<Duration>& y) noexcept;
```

*Returns:* `x.date() == y`.

```template<class Duration>
constexpr bool operator<(const leap_second& x, const sys_time<Duration>& y) noexcept;
```

*Returns:* `x.date() < y`.

```template<class Duration>
constexpr bool operator<=(const leap_second& x, const sys_time<Duration>& y) noexcept;
```

*Returns:* `!(y < x)`.

```template<class Duration>
constexpr bool operator>(const leap_second& x, const sys_time<Duration>& y) noexcept;
```

*Returns:* `!(x < y)`.

```template<class Duration>
constexpr bool operator>=(const leap_second& x, const sys_time<Duration>& y) noexcept;
```

*Returns:* `!(x < y)`.

```template<class Duration>
constexpr bool operator<(const sys_time<Duration>& x, const leap_second& y) noexcept;
```

*Returns:* `!(y < x)`.

```template<class Duration>
constexpr bool operator>=(const sys_time<Duration>& x, const leap_second& y) noexcept;
```

*Returns:* `!(y < x)`.

```template<class Duration>
constexpr bool operator>=(const sys_time<Duration>& x, const leap_second& y) noexcept;
```

*Returns:* `!(x < y)`.

```template<class Duration>
constexpr bool operator>=(const sys_time<Duration>& x, const leap_second& y) noexcept;
```

*Returns:* `!(x < y)`. 

§ 27.11.3
template<class Duration>
constexpr bool operator>=(const sys_time<Duration>& x, const leap_second& y) noexcept;

Returns: !(x < y).

template<class Duration>
requires three_way_comparable_with<sys_seconds, sys_time<Duration>>
constexpr auto operator<=>(const leap_second& x, const sys_time<Duration>& y) noexcept;

Returns: x.date() <=> y.

27.11.9 Class time_zone_link

27.11.9.1 Overview

namespace std::chrono {
    class time_zone_link {
    public:
        time_zone_link(time_zone_link&&) = default;
        time_zone_link& operator=(time_zone_link&&) = default;
        // unspecified additional constructors
        string_view name() const noexcept;
        string_view target() const noexcept;
    };
}

A time_zone_link specifies an alternative name for a time_zone. time_zone_links are constructed when the time zone database is initialized.

27.11.9.2 Member functions

string_view name() const noexcept;

Returns: The alternative name for the time zone.

string_view target() const noexcept;

Returns: The name of the time_zone for which this time_zone_link provides an alternative name.

27.11.9.3 Non-member functions

bool operator==(const time_zone_link& x, const time_zone_link& y) noexcept;

Returns: x.name() == y.name().

strong_ordering operator<=>(const time_zone_link& x, const time_zone_link& y) noexcept;

Returns: x.name() <=> y.name().

27.12 Formatting

Each formatter (20.20.5) specialization in the chrono library (27.2) meets the Formatter requirements (20.20.5.1). The parse member functions of these formatters interpret the format specification as a chrono-format-spec according to the following syntax:

chrono-format-spec:
    fill-and-align\opt width\opt precision\opt chrono-specs\opt
chrono-specs:
    conversion-spec
    chrono-specs conversion-spec
    chrono-specs literal-char

literal-char:
    any character other than {, }, or %
conversion-spec:
    % modifier\opt type
    modifier: one of
    E 0
The productions `fill-and-align`, `width`, and `precision` are described in 20.20.2. Giving a `precision` specification in the `chrono-format-spec` is valid only for `std::chrono::duration` types where the representation type `Rep` is a floating-point type. For all other `Rep` types, an exception of type `format_error` is thrown if the `chrono-format-spec` contains a `precision` specification. All ordinary multibyte characters represented by `literal-char` are copied unchanged to the output.

Each conversion specifier `conversion-spec` is replaced by appropriate characters as described in Table 99; the formats specified in ISO 8601:2004 shall be used where so described. Some of the conversion specifiers depend on the locale that is passed to the formatting function if the latter takes one, or the global locale otherwise. If the formatted object does not contain the information the conversion specifier refers to, an exception of type `format_error` is thrown.

The result of formatting a `std::chrono::duration` instance holding a negative value, or an `hh_mm_ss` object `h` for which `h.is_negative()` is `true`, is equivalent to the output of the corresponding positive value, with a `STATICALLY-WIDEN<charT>("-")` character sequence placed before the replacement of the initial conversion specifier.

[Example 1:]
```
cout << format{"{:%T}"}, -10'000s); // prints: -02:46:40
cout << format{"{:%H:%M:%S}"}, -10'000s); // prints: -02:46:40
cout << format{"minutes {:%M, hours %H, seconds %S}"}, -10'000s); // prints: minutes -46, hours 02, seconds 40
```

—end example—

Unless explicitly requested, the result of formatting a chrono type does not contain time zone abbreviation and time zone offset information. If the information is available, the conversion specifiers `%Z` and `%z` will format this information (respectively).

[Note 1: If the information is not available and a `%Z` or `%z` conversion specifier appears in the `chrono-format-spec`, an exception of type `format_error` is thrown, as described above. —end note—]

If the type being formatted does not contain the information that the format flag needs, an exception of type `format_error` is thrown.

[Example 2: A `duration` does not contain enough information to format as a `weekday`. —end example—]

However, if a flag refers to a “time of day” (e.g. `%H`, `%I`, `%p`, etc.), then a specialization of `duration` is interpreted as the time of day elapsed since midnight.

Table 99: Meaning of conversion specifiers  [tab:time.format.spec]

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%a</code></td>
<td>The locale’s abbreviated weekday name. If the value does not contain a valid weekday, an exception of type <code>format_error</code> is thrown.</td>
</tr>
<tr>
<td><code>%A</code></td>
<td>The locale’s full weekday name. If the value does not contain a valid weekday, an exception of type <code>format_error</code> is thrown.</td>
</tr>
<tr>
<td><code>%b</code></td>
<td>The locale’s abbreviated month name. If the value does not contain a valid month, an exception of type <code>format_error</code> is thrown.</td>
</tr>
<tr>
<td><code>%B</code></td>
<td>The locale’s full month name. If the value does not contain a valid month, an exception of type <code>format_error</code> is thrown.</td>
</tr>
<tr>
<td><code>%c</code></td>
<td>The locale’s date and time representation. The modified command <code>%Ec</code> produces the locale’s alternate date and time representation.</td>
</tr>
<tr>
<td><code>%C</code></td>
<td>The year divided by 100 using floored division. If the result is a single decimal digit, it is prefixed with 0. The modified command <code>%Ec</code> produces the locale’s alternative representation of the century.</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with 0. The modified command <code>%Od</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%D</code></td>
<td>Equivalent to <code>%m/%d/%y</code>.</td>
</tr>
<tr>
<td>Specifier</td>
<td>Replacement</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>%e</td>
<td>The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with a space. The modified command %0e produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%F</td>
<td>Equivalent to %Y-%m-%d.</td>
</tr>
<tr>
<td>%g</td>
<td>The last two decimal digits of the ISO week-based year. If the result is a single digit it is prefixed by 0.</td>
</tr>
<tr>
<td>%G</td>
<td>The ISO week-based year as a decimal number. If the result is less than four digits it is left-padded with 0 to four digits.</td>
</tr>
<tr>
<td>%h</td>
<td>Equivalent to %b.</td>
</tr>
<tr>
<td>%H</td>
<td>The hour (24-hour clock) as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %0H produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%I</td>
<td>The hour (12-hour clock) as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %0I produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%j</td>
<td>If the type being formatted is a specialization of <code>duration</code>, the decimal number of days without padding. Otherwise, the day of the year as a decimal number. Jan 1 is 001. If the result is less than three digits, it is left-padded with 0 to three digits.</td>
</tr>
<tr>
<td>%m</td>
<td>The month as a decimal number. Jan is 01. If the result is a single digit, it is prefixed with 0. The modified command %0m produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%M</td>
<td>The minute as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %0M produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%n</td>
<td>A new-line character.</td>
</tr>
<tr>
<td>%p</td>
<td>The locale’s equivalent of the AM/PM designations associated with a 12-hour clock.</td>
</tr>
<tr>
<td>%q</td>
<td>The duration’s unit suffix as specified in 27.5.11.</td>
</tr>
<tr>
<td>%Q</td>
<td>The duration’s numeric value (as if extracted via <code>.count()</code>).</td>
</tr>
<tr>
<td>%r</td>
<td>The locale’s 12-hour clock time.</td>
</tr>
<tr>
<td>%R</td>
<td>Equivalent to %H:%M.</td>
</tr>
<tr>
<td>%s</td>
<td>Seconds as a decimal number. If the number of seconds is less than 10, the result is prefixed with 0. If the precision of the input cannot be exactly represented with seconds, then the format is a decimal floating-point number with a fixed format and a precision matching that of the precision of the input (or to a microseconds precision if the conversion to floating-point decimal seconds cannot be made within 18 fractional digits). The character for the decimal point is localized according to the locale. The modified command %0S produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%t</td>
<td>A horizontal-tab character.</td>
</tr>
<tr>
<td>%T</td>
<td>Equivalent to %H:%M:%S.</td>
</tr>
<tr>
<td>%u</td>
<td>The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command %0u produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%U</td>
<td>The week number of the year as a decimal number. The first Sunday of the year is the first day of week 01. Days of the same year prior to that are in week 00. If the result is a single digit, it is prefixed with 0. The modified command %0U produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%V</td>
<td>The ISO week-based week number as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %0V produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%w</td>
<td>The weekday as a decimal number (0-6), where Sunday is 0. The modified command %0w produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%W</td>
<td>The ISO week-based week number as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %0W produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%x</td>
<td>The locale’s date representation. The modified command %Ex produces the locale’s alternate date representation.</td>
</tr>
</tbody>
</table>
Table 99: Meaning of conversion specifiers (continued)

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>%X</td>
<td>The locale’s time representation. The modified command %EX produces the locale’s alternate time representation.</td>
</tr>
<tr>
<td>%y</td>
<td>The last two decimal digits of the year. If the result is a single digit it is prefixed by 0. The modified command %Ey produces the locale’s alternative representation. The modified command %EC produces the locale’s alternative representation of offset from year only.</td>
</tr>
<tr>
<td>%Y</td>
<td>The year as a decimal number. If the result is less than four digits it is left-padded with 0 to four digits. The modified command %EY produces the locale’s alternative full year representation.</td>
</tr>
<tr>
<td>%z</td>
<td>The offset from UTC in the ISO 8601:2004 format. For example -0430 refers to 4 hours 30 minutes behind UTC. If the offset is zero, +0000 is used. The modified commands %Ez and %Oz insert a : between the hours and minutes: -04:30. If the offset information is not available, an exception of type format_error is thrown.</td>
</tr>
<tr>
<td>%Z</td>
<td>The time zone abbreviation. If the time zone abbreviation is not available, an exception of type format_error is thrown.</td>
</tr>
<tr>
<td>%%</td>
<td>A % character.</td>
</tr>
</tbody>
</table>

6 If the chrono-specs is omitted, the chrono object is formatted as if by streaming it to std::stringstream os and copying os.str() through the output iterator of the context with additional padding and adjustments as specified by the format specifiers.

[Example 3:]
```cpp
string s = format("{:=>8}", 42ms); // value of s is "====42ms"
end example]
```
template<class Duration, class charT>
struct formatter<chrono::file_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with \texttt{STATICALLY-WIDEN}<\texttt{charT}>("UTC"). If %z (or a modified variant of %z) is used, an offset of 0min is formatted. The date and time formatted are equivalent to those formatted by a \texttt{sys_time} initialized with \texttt{clock_cast<system_clock>(t)}, or by a \texttt{utc_time} initialized with \texttt{clock_cast<utc_clock>(t)}, where \(t\) is the first argument to \texttt{format}.

template<class Duration, class charT>
struct formatter<chrono::local_time<Duration>, charT>;

Remarks: If %Z, %z, or a modified version of %z is used, an exception of type \texttt{format_error} is thrown.

\begin{verbatim}
template<class Duration> struct local_time_format_t {
  // exposition only
  local_time<Duration> time;
  // exposition only
  const string* abbrev;
  // exposition only
  const seconds* offset_sec;
};

template<class Duration>
local_time_format_t<Duration> local_time_format(local_time<Duration> time, const string* abbrev = nullptr,
const seconds* offset_sec = nullptr);
\end{verbatim}

Returns: \{time, abbrev, offset_sec\}.

Let \(f\) be a \texttt{local-time-format-t<Duration>} object passed to \texttt{formatter::format}.

Remarks: If %Z is used, it is replaced with \(*f.abbrev\) if \(f.abbrev\) is not a null pointer value. If %Z is used and \(f.abbrev\) is a null pointer value, an exception of type \texttt{format_error} is thrown. If %z (or a modified variant of %z) is used, it is formatted with the value of \(*f.offset_sec\) if \(f.offset_sec\) is not a null pointer value. If %z (or a modified variant of %z) is used and \(f.offset_sec\) is a null pointer value, then an exception of type \texttt{format_error} is thrown.

\begin{verbatim}
template<class FormatContext>
type_name FormatContext::iterator
  format(const chrono::zoned_time<Duration, TimeZonePtr>& tp, FormatContext& ctx);
\end{verbatim}

Effects: Equivalent to:

\begin{verbatim}
sys_info info = tp.get_info();
return format(const chrono::local_time<Duration, TimeZonePtr>& tp, FormatContext& ctx);
\end{verbatim}

\subsection{27.13 Parsing}\hspace{1cm} [time.parse]

Each \texttt{parse} overload specified in this subclause calls \texttt{from_stream} unqualified, so as to enable argument dependent lookup (6.5.3). In the following paragraphs, let \(is\) denote an object of type \texttt{basic_istream<charT, traits>} and let \(I\) denote an object of type \texttt{basic_istream<charT, traits>}&, where \texttt{charT} and \texttt{traits} are template parameters in that context.

\begin{verbatim}
template<class charT, class traits, class Alloc, class Parsable>
unspecified
  parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp);
\end{verbatim}

Constraints: The expression

\begin{verbatim}
from_stream(declval<basic_istream<charT, traits>&>(), fmt.c_str(), tp)
\end{verbatim}

is well-formed when treated as an unevaluated operand.
Returns: A manipulator such that the expression \( \text{is} \ >> \text{parse}(\text{fmt}, \text{tp}) \) has type \( I \), has value \( \text{is} \), and calls \( \text{from}\_\text{stream}(\text{is}, \text{fmt}\_.\text{c}\_\text{str}(), \text{tp}) \).

```cpp
template<class charT, class traits, class Alloc, class Parsable>
unspec	parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp,
          basic_string<charT, traits, Alloc>& abbrev);
```

Constraints: The expression
\[
\text{from}\_\text{stream}(\text{declval<basic\_istream}<\text{charT, traits>&>()}, \text{fmt}\_.\text{c}\_\text{str}(), \text{tp}, \text{addressof(abbrev)})
\]

is well-formed when treated as an unevaluated operand.

Returns: A manipulator such that the expression \( \text{is} \ >> \text{parse}(\text{fmt, tp, abbrev}) \) has type \( I \), has value \( \text{is} \), and calls \( \text{from}\_\text{stream}(\text{is}, \text{fmt}\_.\text{c}\_\text{str}(), \text{tp}, \text{addressof(abbrev)}) \).

```cpp
template<class charT, class traits, class Alloc, class Parsable>
unspec	parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp,
          minutes& offset);
```

Constraints: The expression
\[
\text{from}\_\text{stream}(\text{declval<basic\_istream}<\text{charT, traits>&>()}, \text{fmt}\_.\text{c}\_\text{str}(), \text{tp},
          \text{declval<basic\_string<charT, traits, Alloc>>&>()}, \&offset)
\]

is well-formed when treated as an unevaluated operand.

Returns: A manipulator such that the expression \( \text{is} \ >> \text{parse}(\text{fmt, tp, offset}) \) has type \( I \), has value \( \text{is} \), and calls:
\[
\text{from}\_\text{stream}(\text{is}, \text{fmt}\_.\text{c}\_\text{str}(), \text{tp},
          \text{static}\_\text{cast<basic\_string<charT, traits, Alloc>>&>()}(\text{nullptr}), \&offset)
\]

```cpp
template<class charT, class traits, class Alloc, class Parsable>
unspec	parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp,
          basic_string<charT, traits, Alloc>& abbrev, minutes& offset);
```

Constraints: The expression
\[
\text{from}\_\text{stream}(\text{declval<basic\_istream}<\text{charT, traits>&>()}, \text{fmt}\_.\text{c}\_\text{str}(), \text{tp},
          \text{addressof(abbrev)}, \&offset)
\]

is well-formed when treated as an unevaluated operand.

Returns: A manipulator such that the expression \( \text{is} \ >> \text{parse}(\text{fmt, tp, abbrev, offset}) \) has type \( I \), has value \( \text{is} \), and calls \( \text{from}\_\text{stream}(\text{is}, \text{fmt}\_.\text{c}\_\text{str}(), \text{tp}, \text{addressof(abbrev)}, \&offset) \).

All \text{from}\_\text{stream} overloads behave as unformatted input functions, except that they have an unspecified effect on the value returned by subsequent calls to \text{basic\_istream}::<\text{gcount}>(). Each overload takes a format string containing ordinary characters and flags which have special meaning. Each flag begins with a \( \% \). Some flags can be modified by \( E \) or \( O \). During parsing each flag interprets characters as parts of date and time types according to Table 100. Some flags can be modified by a width parameter given as a positive decimal integer called out as \( N \) below which governs how many characters are parsed from the stream in interpreting the flag. All characters in the format string that are not represented in Table 100, except for whitespace, are parsed unchanged from the stream. A whitespace character matches zero or more whitespace characters in the input stream.

If the type being parsed cannot represent the information that the format flag refers to, \( \text{is}.\text{setstate(ios\_base::failbit)} \) is called.

[Example 1: A duration cannot represent a weekday. — end example]  
However, if a flag refers to a “time of day” (e.g. \( \%H, \%I, \%p \), etc.), then a specialization of \text{duration} is parsed as the time of day elapsed since midnight.

§ 27.13
If the `from_stream` overload fails to parse everything specified by the format string, or if insufficient information is parsed to specify a complete duration, time point, or calendrical data structure, `setstate(ios_base::failbit)` is called on the `basic_istream`.

### Table 100: Meaning of parse flags [tab:time.parse.spec]

<table>
<thead>
<tr>
<th>Flag</th>
<th>Parsed value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%a</code></td>
<td>The locale’s full or abbreviated case-insensitive weekday name.</td>
</tr>
<tr>
<td><code>%A</code></td>
<td>Equivalent to <code>%a</code>.</td>
</tr>
<tr>
<td><code>%b</code></td>
<td>The locale’s full or abbreviated case-insensitive month name.</td>
</tr>
<tr>
<td><code>%B</code></td>
<td>Equivalent to <code>%b</code>.</td>
</tr>
<tr>
<td><code>%c</code></td>
<td>The locale’s date and time representation. The modified command <code>%Ec</code> interprets the locale’s alternate date and time representation.</td>
</tr>
<tr>
<td><code>%C</code></td>
<td>The century as a decimal number. The modified command <code>%NC</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command <code>%EC</code> interprets the locale’s alternative representation of the century.</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>The day of the month as a decimal number. The modified command <code>%Nd</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command <code>%Od</code> interprets the locale’s alternative representation of the day of the month.</td>
</tr>
<tr>
<td><code>%D</code></td>
<td>Equivalent to <code>%m/%d/%y</code>.</td>
</tr>
<tr>
<td><code>%e</code></td>
<td>Equivalent to <code>%d</code> and can be modified like <code>%d</code>.</td>
</tr>
<tr>
<td><code>%f</code></td>
<td>Equivalent to <code>%Y-%m-%d</code>. If modified with a width <code>N</code>, the width is applied to only <code>%Y</code>.</td>
</tr>
<tr>
<td><code>%g</code></td>
<td>The last two decimal digits of the ISO week-based year. The modified command <code>%Ng</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td><code>%G</code></td>
<td>The ISO week-based year as a decimal number. The modified command <code>%NG</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 4. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td><code>%h</code></td>
<td>Equivalent to <code>%b</code>.</td>
</tr>
<tr>
<td><code>%H</code></td>
<td>The hour (24-hour clock) as a decimal number. The modified command <code>%NH</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command <code>%OH</code> interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%I</code></td>
<td>The hour (12-hour clock) as a decimal number. The modified command <code>%NI</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command <code>%OI</code> interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%j</code></td>
<td>If the type being parsed is a specialization of <code>duration</code>, a decimal number of <code>days</code>. Otherwise, the day of the year as a decimal number. Jan 1 is 1. In either case, the modified command <code>%Nj</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 3. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td><code>%m</code></td>
<td>The month as a decimal number. Jan is 1. The modified command <code>%Nm</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command <code>%Om</code> interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%M</code></td>
<td>The minutes as a decimal number. The modified command <code>%Nm</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command <code>%OM</code> interprets the locale’s alternative representation.</td>
</tr>
</tbody>
</table>
| `%n` | Matches one whitespace character.  
  [Note 1: `%n`, `%t`, and a space can be combined to match a wide range of whitespace patterns. For example, "%n " matches one or more whitespace characters, and "%n%t%t" matches one to three whitespace characters. — end note] |
| `%p` | The locale’s equivalent of the AM/PM designations associated with a 12-hour clock. |
| `%r` | The locale’s 12-hour clock time. |
| `%s` | Equivalent to `%H:%M`. |
Table 100: Meaning of parse flags (continued)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Parsed value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%S</code></td>
<td>The seconds as a decimal number. The modified command <code>%NS</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2 if the input time has a precision convertible to seconds. Otherwise the default width is determined by the decimal precision of the input and the field is interpreted as a <code>long double</code> in a fixed format. If encountered, the locale determines the decimal point character. Leading zeroes are permitted but not required. The modified command <code>%OS</code> interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%t</code></td>
<td>Matches zero or one whitespace characters.</td>
</tr>
<tr>
<td><code>%T</code></td>
<td>Equivalent to <code>%H:%M:%S</code>.</td>
</tr>
<tr>
<td><code>%u</code></td>
<td>The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command <code>%Nu</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 1. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td><code>%U</code></td>
<td>The week number of the year as a decimal number. The first Sunday of the year is the first day of week 01. Days of the same year prior to that are in week 00. The modified command <code>%NU</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command <code>%OU</code> interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%V</code></td>
<td>The ISO week-based week number as a decimal number. The modified command <code>%NV</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td><code>%w</code></td>
<td>The weekday as a decimal number (0-6), where Sunday is 0. The modified command <code>%Nw</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 1. Leading zeroes are permitted but not required. The modified command <code>%Ow</code> interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%W</code></td>
<td>The week number of the year as a decimal number. The first Monday of the year is the first day of week 01. Days of the same year prior to that are in week 00. The modified command <code>%NW</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required. The modified commands <code>%OW</code> and <code>%Oy</code> interpret the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%x</code></td>
<td>The locale’s date representation. The modified command <code>%Ex</code> interprets the locale’s alternate date representation.</td>
</tr>
<tr>
<td><code>%X</code></td>
<td>The locale’s time representation. The modified command <code>%EX</code> interprets the locale’s alternate time representation.</td>
</tr>
<tr>
<td><code>%y</code></td>
<td>The last two decimal digits of the year. If the century is not otherwise specified (e.g. with <code>%C</code>), values in the range [69, 99] are presumed to refer to the years 1969 to 1999, and values in the range [00, 68] are presumed to refer to the years 2000 to 2068. The modified command <code>%Ny</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 2. Leading zeroes are permitted but not required. The modified commands <code>%Ey</code> and <code>%Oy</code> interpret the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%Y</code></td>
<td>The year as a decimal number. The modified command <code>%NY</code> specifies the maximum number of characters to read. If <code>N</code> is not specified, the default is 4. Leading zeroes are permitted but not required. The modified command <code>%EY</code> interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%z</code></td>
<td>The offset from UTC in the format `[+</td>
</tr>
<tr>
<td><code>%Z</code></td>
<td>The time zone abbreviation or name. A single word is parsed. This word can only contain characters from the basic source character set (5.3) that are alphanumeric, or one of '-', '/', '-', or '+'.</td>
</tr>
<tr>
<td><code>%%</code></td>
<td>A <code>%</code> character is extracted.</td>
</tr>
</tbody>
</table>


27.14 Header <ctime> synopsis

```cpp
namespace std {
    using size_t = see 17.2.4;
    using clock_t = see below;
    using time_t = see below;

    struct timespec;
    struct tm;

    clock_t clock();
    double difftime(time_t time1, time_t time0);
    time_t mktime(struct tm* timeptr);
    time_t time(time_t* timer);
    int timespec_get(timespec* ts, int base);
    char* asctime(const struct tm* timeptr);
    char* ctime(const time_t* timer);
    struct tm* gmtime(const time_t* timer);
    struct tm* localtime(const time_t* timer);
    size_t strftime(char* s, size_t maxsize, const char* format, const struct tm* timeptr);
}
```

1 The contents of the header <ctime> are the same as the C standard library header <time.h>.

2 The functions asctime, ctime, gmtime, and localtime are not required to avoid data races (16.4.6.10).

See also: ISO C 7.27

---

261) strftime supports the C conversion specifiers C, D, e, F, g, G, h, r, R, t, T, u, V, and z, and the modifiers E and D.
28  Localization library

28.1  General

This Clause describes components that C++ programs may use to encapsulate (and therefore be more portable when confronting) cultural differences. The locale facility includes internationalization support for character classification and string collation, numeric, monetary, and date/time formatting and parsing, and message retrieval.

The following subclauses describe components for locales themselves, the standard facets, and facilities from the ISO C library, as summarized in Table 101.

Table 101: Localization library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.3</td>
<td>locales &lt;locale&gt;</td>
</tr>
<tr>
<td>28.4</td>
<td>Standard locale categories &lt;locale&gt;</td>
</tr>
<tr>
<td>28.5</td>
<td>C library locales &lt;locale&gt;</td>
</tr>
</tbody>
</table>

28.2  Header <locale> synopsis

namespace std {
    // 28.3.1, locale
    class locale;
    template<class Facet> const Facet& use_facet(const locale&);
    template<class Facet> bool has_facet(const locale&) noexcept;

    // 28.3.3, convenience interfaces
    template<class charT> bool isspace (charT c, const locale& loc);
    template<class charT> bool isprint (charT c, const locale& loc);
    template<class charT> bool iscntrl (charT c, const locale& loc);
    template<class charT> bool isupper (charT c, const locale& loc);
    template<class charT> bool islower (charT c, const locale& loc);
    template<class charT> bool isalpha (charT c, const locale& loc);
    template<class charT> bool isdigit (charT c, const locale& loc);
    template<class charT> bool ispunct (charT c, const locale& loc);
    template<class charT> bool isxdigit(charT c, const locale& loc);
    template<class charT> bool isalnum (charT c, const locale& loc);
    template<class charT> bool isgraph (charT c, const locale& loc);
    template<class charT> bool isblank (charT c, const locale& loc);
    template<class charT> charT toupper(charT c, const locale& loc);
    template<class charT> charT tolower(charT c, const locale& loc);

    // 28.4.2, ctype
    class ctype_base;
    template<class charT> class ctype;
    template<> class ctype<char>;
    template<class charT> class ctype_byname; // specialization
    class codecvt_base;
    template<class internT, class externT, class stateT> class codecvt;
    template<class internT, class externT, class stateT> class codecvt_byname;

    // 28.4.3, numeric
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class num_get;
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class num_put;
    template<class charT>
    class numpunct;
}
template<class charT>
    class numpunct_byname;

// 28.4.5, collation
    template<class charT> class collate;
    template<class charT> class collate_byname;

// 28.4.6, date and time
    class time_base;
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
        class time_get;
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
        class time_get_byname;
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
        class time_put;
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
        class time_put_byname;

// 28.4.7, money
    class money_base;
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
        class money_get;
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
        class money_put;
    template<class charT, bool Intl = false>
        class moneypunct;
    template<class charT, bool Intl = false>
        class moneypunct_byname;

// 28.4.8, message retrieval
    class messages_base;
    template<class charT> class messages;
    template<class charT> class messages_byname;
}

1 The header `<locale>` defines classes and declares functions that encapsulate and manipulate the information peculiar to a locale.\textsuperscript{262}

\section{Locales} \label{locales}

\subsection{Class locale} \label{locale}

\subsubsection{General} \label{locale.general}

namespace std {
    class locale {
    public:
        // types
        class facet;
        class id;
        using category = int;
        static const category // values assigned here are for exposition only
            none = 0,
            collate = 0x010, ctype = 0x020,
            monetary = 0x040, numeric = 0x080,
            time = 0x100, messages = 0x200,
            all = collate | ctype | monetary | numeric | time | messages;

        // construct/copy/destroy
        locale() noexcept;
        locale(const locale& other) noexcept;
        explicit locale(const char* std_name);
        explicit locale(const string& std_name);
        locale(const locale& other, const char* std_name, category);

\end{verbatim}

\textsuperscript{262} In this subclause, the type name \texttt{struct tm} is an incomplete type that is defined in \texttt{<ctime> (27.14).}
locale(const locale& other, const string& std_name, category);
locale(const locale& other, Facet* f);
locale(const locale& other, const locale& one, category);
~locale();  // not virtual
const locale& operator=(const locale& other) noexcept;
locale combine(const locale& other) const;

// locale operations
string name() const;

bool operator==(const locale& other) const;

template<class charT, class traits, class Allocator>
bool operator()(const basic_string<charT, traits, Allocator>& s1,
               const basic_string<charT, traits, Allocator>& s2) const;

// global locale objects
static locale global(const locale&);
static const locale& classic();

1 Class locale implements a type-safe polymorphic set of facets, indexed by facet type. In other words, a facet has a dual role: in one sense, it’s just a class interface; at the same time, it’s an index into a locale’s set of facets.

2 Access to the facets of a locale is via two function templates, use_facet<> and has_facet<>.

3 [Example 1: An iostream operator<< might be implemented as:]

4 In the call to use_facet<Facet>(loc), the type argument chooses a facet, making available all members of the named type. If Facet is not present in a locale, it throws the standard exception bad_cast. A C++ program can check if a locale implements a particular facet with the function template has_facet<Facet>(). User-defined facets may be installed in a locale, and used identically as may standard facets.

5 [Note 1: All locale semantics are accessed via use_facet<> and has_facet<>., except that:

6 Once a facet reference is obtained from a locale object by calling use_facet<>., that reference remains usable, and the results from member functions of it may be cached and re-used, as long as some locale object refers to that facet.

263) Note that in the call to put, the stream is implicitly converted to an ostreambuf_iterator<charT, traits>.

§ 28.3.1.1
In successive calls to a locale facet member function on a facet object installed in the same locale, the returned result shall be identical.

8 A locale constructed from a name string (such as "POSIX"), or from parts of two named locales, has a name; all others do not. Named locales may be compared for equality; an unnamed locale is equal only to (copies of) itself. For an unnamed locale, `locale::name()` returns the string "*".

9 Whether there is one global locale object for the entire program or one global locale object per thread is implementation-defined. Implementations should provide one global locale object per thread. If there is a single global locale object for the entire program, implementations are not required to avoid data races on it (16.4.6.10).

28.3.1.2 Types

28.3.1.2.1 Type `locale::category`

```cpp
using category = int;
```

1 Valid category values include the `locale` member bitmask elements `collate`, `ctype`, `monetary`, `numeric`, `time`, and `messages`, each of which represents a single locale category. In addition, `locale` member bitmask constant `none` is defined as zero and represents no category. And `locale` member bitmask constant `all` is defined such that the expression

```
(collate | ctype | monetary | numeric | time | messages | all) == all
```

is true, and represents the union of all categories. Further, the expression `(X | Y)`, where `X` and `Y` each represent a single category, represents the union of the two categories.

2 `locale` member functions expecting a `category` argument require one of the `category` values defined above, or the union of two or more such values. Such a `category` value identifies a set of locale categories. Each locale category, in turn, identifies a set of locale facets, including at least those shown in Table 102.

### Table 102: Locale category facets

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td><code>collate&lt;char&gt;</code>, <code>collate&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>ctype</td>
<td><code>ctype&lt;char&gt;</code>, <code>ctype&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt&lt;char, char, mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt&lt;char16_t, char8_t, mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt&lt;char32_t, char8_t, mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt&lt;wchar_t, char, mbstate_t&gt;</code></td>
</tr>
<tr>
<td>monetary</td>
<td><code>moneypunct&lt;char&gt;</code>, <code>moneypunct&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>moneypunct&lt;char, true&gt;</code>, <code>moneypunct&lt;wchar_t, true&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>money_get&lt;char&gt;</code>, <code>money_get&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>money_put&lt;char&gt;</code>, <code>money_put&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>numeric</td>
<td><code>numpunct&lt;char&gt;</code>, <code>numpunct&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>num_get&lt;char&gt;</code>, <code>num_get&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>num_put&lt;char&gt;</code>, <code>num_put&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>time</td>
<td><code>time_get&lt;char&gt;</code>, <code>time_get&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>time_put&lt;char&gt;</code>, <code>time_put&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>messages</td>
<td><code>messages&lt;char&gt;</code>, <code>messages&lt;wchar_t&gt;</code></td>
</tr>
</tbody>
</table>

For any locale `loc` either constructed, or returned by `locale::classic()`, and any facet `Facet` shown in Table 102, `has_facet<Facet>(loc)` is true. Each `locale` member function which takes a `locale::category` argument operates on the corresponding set of facets.

An implementation is required to provide those specializations for facet templates identified as members of a category, and for those shown in Table 103.

The provided implementation of members of facets `num_get<charT>` and `num_put<charT>` calls `use_facet<F>(1)` only for facet `F` of types `numpunct<charT>` and `ctype<charT>`, and for locale `l` the value obtained by calling member `getloc()` on the `ios_base&` argument to these functions.

In declarations of facets, a template parameter with name `InputIterator` or `OutputIterator` indicates the set of all possible specializations on parameters that meet the `Cpp17InputIterator` requirements or
Table 103: Required specializations  [tab:locale.spec]

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td>collate_byname&lt;char&gt;, collate_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>ctype</td>
<td>ctype_byname&lt;char&gt;, ctype_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char16_t, char8_t, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char32_t, char8_t, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;wchar_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td>monetary</td>
<td>moneypunct_byname&lt;char, International&gt;</td>
</tr>
<tr>
<td></td>
<td>moneypunct_byname&lt;wchar_t, International&gt;</td>
</tr>
<tr>
<td></td>
<td>money_get&lt;C, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>money_put&lt;C, OutputIterator&gt;</td>
</tr>
<tr>
<td>numeric</td>
<td>numpunct_byname&lt;char&gt;, numpunct_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_get&lt;C, InputIterator&gt;, num_put&lt;C, OutputIterator&gt;</td>
</tr>
<tr>
<td>time</td>
<td>time_get&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get_byname&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get_byname&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put_byname&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put_byname&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td>messages</td>
<td>messages_byname&lt;char&gt;, messages_byname&lt;wchar_t&gt;</td>
</tr>
</tbody>
</table>

Cpp17OutputIterator requirements, respectively (23.3). A template parameter with name C represents the set of types containing char, wchar_t, and any other implementation-defined character types that meet the requirements for a character on which any of the iostream components can be instantiated. A template parameter with name International represents the set of all possible specializations on a bool parameter.

28.3.1.2.2 Class locale::facet  [locale.facet]

namespace std {
    class locale::facet {
        protected:
            explicit facet(size_t refs = 0);  
            virtual ~facet();
            facet(const facet&) = delete;
            void operator=(const facet&) = delete;
        }
    }

1 Class facet is the base class for locale feature sets. A class is a facet if it is publicly derived from another facet, or if it is a class derived from locale::facet and contains a publicly accessible declaration as follows:\textsuperscript{264}

static ::std::locale::id id;

2 Template parameters in this Clause which are required to be facets are those named Facet in declarations. A program that passes a type that is not a facet, or a type that refers to a volatile-qualified facet, as an (explicit or deduced) template parameter to a locale function expecting a facet, is ill-formed. A const-qualified facet is a valid template argument to any locale function that expects a Facet template parameter.

3 The refs argument to the constructor is used for lifetime management. For refs == 0, the implementation performs delete static_cast<locale::facet*>(f) (where f is a pointer to the facet) when the last locale object containing the facet is destroyed; for refs == 1, the implementation never destroys the facet.

4 Constructors of all facets defined in this Clause take such an argument and pass it along to their facet base class constructor. All one-argument constructors defined in this Clause are explicit, preventing their participation in automatic conversions.

\textsuperscript{264} This is a complete list of requirements; there are no other requirements. Thus, a facet class need not have a public copy constructor, assignment, default constructor, destructor, etc.

§ 28.3.1.2.2 1339
For some standard facets a standard “...byname” class, derived from it, implements the virtual function semantics equivalent to that facet of the locale constructed by `locale(const char*)` with the same name. Each such facet provides a constructor that takes a `const char*` argument, which names the locale, and a `refs` argument, which is passed to the base class constructor. Each such facet also provides a constructor that takes a `string` argument `str` and a `refs` argument, which has the same effect as calling the first constructor with the two arguments `str.c_str()` and `refs`. If there is no “...byname” version of a facet, the base class implements named locale semantics itself by reference to other facets.

28.3.1.2.3 Class `locale::id`

```cpp
namespace std {
    class locale::id {
        public:
            id();
            void operator=(const id&) = delete;
            id(const id&) = delete;
    };
}
```

The class `locale::id` provides identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.

2 [Note 1: Because facets are used by iostreams, potentially while static constructors are running, their initialization cannot depend on programmed static initialization. One initialization strategy is for `locale` to initialize each facet’s `id` member the first time an instance of the facet is installed into a locale. This depends only on static storage being zero before constructors run (6.9.3.2). — end note]

28.3.1.3 Constructors and destructor

```cpp
locale() noexcept;

explicit locale(const char* std_name);

explicit locale(const string& std_name);

locale(const locale& other, const char* std_name, category);

locale(const locale& other, const string& std_name, category cat);

template<class Facet> locale(const locale& other, Facet* f);
```

Effects: Constructs a copy of the argument last passed to `locale::global(locale&)`, if it has been called; else, the resulting facets have virtual function semantics identical to those of `locale::classic()`.

[Note 1: This constructor yields a copy of the current global locale. It is commonly used as a default argument for function parameters of type `const locale&`. — end note]

```cpp
explicit locale(const char* std_name);

locale(const locale& other, const char* std_name, category);

locale(const locale& other, const string& std_name, category cat);

template<class Facet> locale(const locale& other, Facet* f);
```

Effects: Constructs a locale using standard C locale names, e.g., "POSIX". The resulting locale implements semantics defined to be associated with that name.

Throws: `runtime_error` if the argument is not valid, or is null.

Remarks: The set of valid string argument values is "C", "", and any implementation-defined values.

```cpp
explicit locale(const string& std_name);
```
locale(const locale& other, const locale& one, category cats);

Effects: Constructs a locale incorporating all facets from the first argument except those that implement cats, which are instead incorporated from the second argument.

Remarks: The resulting locale has a name if and only if the first two arguments have names.

const locale& operator=(const locale& other) noexcept;

Effects: Creates a copy of other, replacing the current value.

Returns: *this.

28.3.1.4 Members

template<class Facet> locale combine(const locale& other) const;

Effects: Constructs a locale incorporating all facets from *this except for that one facet of other that is identified by Facet.

Returns: The newly created locale.

Throws: runtime_error if has_facet<Facet>(other) is false.

Remarks: The resulting locale has no name.

string name() const;

Returns: The name of *this, if it has one; otherwise, the string "*".

28.3.1.5 Operators

bool operator==(const locale& other) const;

Returns: true if both arguments are the same locale, or one is a copy of the other, or each has a name and the names are identical; false otherwise.

template<class charT, class traits, class Allocator>
bool operator()(const basic_string<charT, traits, Allocator>& s1,
const basic_string<charT, traits, Allocator>& s2) const;

Effects: Compares two strings according to the collate<charT> facet.

Returns:

use_facet<collate<charT>>(*this).compare(s1.data(), s1.data() + s1.size(),
s2.data(), s2.data() + s2.size()) < 0

Remarks: This member operator template (and therefore locale itself) meets the requirements for a comparator predicate template argument (Clause 25) applied to strings.

[Example 1: A vector of strings v can be collated according to collation rules in locale loc simply by (25.8.2, 22.3.11):

std::sort(v.begin(), v.end(), loc);
—end example]

28.3.1.6 Static members

static locale global(const locale& loc);

Effects: Sets the global locale to its argument. Causes future calls to the constructor locale() to return a copy of the argument. If the argument has a name, does

setlocale(LC_ALL, loc.name().c_str());

otherwise, the effect on the C locale, if any, is implementation-defined.

Returns: The previous value of locale().

Remarks: No library function other than locale::global() affects the value returned by locale().

[Note 1: See 28.5 for data race considerations when setlocale is invoked. — end note]
static const locale& classic();

Returns: A locale that implements the classic "C" locale semantics, equivalent to the value locale("C").
Remarks: This locale, its facets, and their member functions, do not change with time.

28.3.2 locale globals

```cpp
template<class Facet> const Facet& use_facet(const locale& loc);

Mandates: Facet is a facet class whose definition contains the public static member id as defined in 28.3.1.2.2.
Returns: A reference to the corresponding facet of loc, if present.
Throws: bad_cast if has_facet<Facet>(loc) is false.
Remarks: The reference returned remains valid at least as long as any copy of loc exists.
```

```cpp
template<class Facet> bool has_facet(const locale& loc) noexcept;

Returns: true if the facet requested is present in loc; otherwise false.
```

28.3.3 Convenience interfaces

28.3.3.1 Character classification

```cpp
template<class charT> bool isspace (charT c, const locale& loc);
```

Each of these functions is\(\text{F}\) returns the result of the expression:

\[
\text{use_facet<ctype<charT>>(loc).is(ctype_base::F, c)}
\]

where \(\text{F}\) is the \(\text{ctype_base::mask}\) value corresponding to that function (28.4.2).\(^{265}\)

28.3.3.2 Character conversions

```cpp
template<class charT> charT toupper(charT c, const locale& loc);
```

```cpp
template<class charT> charT tolower(charT c, const locale& loc);
```

Each such member function takes an \(\text{ios_base}\&\) argument whose members flags(), precision(), and width(), specify the format of the corresponding datum (29.5.3). Those functions which need to use other facets call its member getloc() to retrieve the locale imbued there. Formatting facets use the character argument fill to fill out the specified width where necessary.

The \(\text{put()}\) members make no provision for error reporting. (Any failures of the OutputIterator argument can be extracted from the returned iterator.) The \(\text{get()}\) members take an \(\text{ios_base::iostate}\&\) argument whose value they ignore, but set to \(\text{ios_base::failbit}\) in case of a parse error.

\(^{265}\) When used in a loop, it is faster to cache the \(\text{ctype}<\) facet and use it directly, or use the vector form of \(\text{ctype}<::\text{is}.\)
Within subclause 28.4 it is unspecified whether one virtual function calls another virtual function.

### 28.4.2 The `ctype` category

#### 28.4.2.1 General

```cpp
namespace std {
    class ctype_base {
    public:
        using mask = see below;

        // numeric values are for exposition only.
        static const mask space = 1 << 0;
        static const mask print = 1 << 1;
        static const mask cntrl = 1 << 2;
        static const mask upper = 1 << 3;
        static const mask lower = 1 << 4;
        static const mask alpha = 1 << 5;
        static const mask digit = 1 << 6;
        static const mask punct = 1 << 7;
        static const mask xdigit = 1 << 8;
        static const mask blank = 1 << 9;
        static const mask alnum = alpha | digit;
        static const mask graph = alnum | punct;
    }
}
```

1. The type `mask` is a bitmask type (16.3.3.3.4).

### 28.4.2.2 Class template `ctype`

#### 28.4.2.2.1 General

```cpp
namespace std {
    template<class charT>
    class ctype : public locale::facet, public ctype_base {
    public:
        using char_type = charT;
        explicit ctype(size_t refs = 0);

        bool is(mask m, charT c) const;
        const charT* is(const charT* low, const charT* high, mask* vec) const;
        const charT* scan_is(mask m, const charT* low, const charT* high) const;
        const charT* scan_not(mask m, const charT* low, const charT* high) const;
        charT toupper(charT c) const;
        const charT* toupper(charT* low, const charT* high) const;
        charT tolower(charT c) const;
        const charT* tolower(charT* low, const charT* high) const;
        charT widen(char c) const;
        const char* widen(const char* low, const char* high, charT* to) const;
        char narrow(charT c, char dfault) const;
        const charT* narrow(const charT* low, const charT* high, char dfault, char* to) const;

        static locale::id id;
    protected:
        ~ctype();
        virtual bool do_is(mask m, charT c) const;
        virtual const charT* do_is(const charT* low, const charT* high, mask* vec) const;
        virtual const charT* do_scan_is(mask m, const charT* low, const charT* high) const;
        virtual const charT* do_scan_not(mask m, const charT* low, const charT* high) const;
        virtual charT do_toupper(charT) const;
        virtual const charT* do_toupper(charT* low, const charT* high) const;
        virtual charT do_toluter(charT) const;
        virtual const charT* do_toluter(charT* low, const charT* high) const;
    }
}
```
virtual charT do_widen(char) const;
virtual const char* do_widen(const char* low, const char* high, charT* dest) const;
virtual char do_narrow(charT, char dfault) const;
virtual const charT* do_narrow(const charT* low, const charT* high,
     char dfault, charT* dest) const;
};
}

1 Class ctype encapsulates the C library <cctype> features. istream members are required to use ctype<> for character classing during input parsing.

2 The specializations required in Table 102 (28.3.1.2.1), namely ctype<char> and ctype<wchar_t>, implement character classing appropriate to the implementation’s native character set.

28.4.2.2.2 ctype members

bool     is(mask m, charT c) const;
const charT* is(const charT* low, const charT* high, mask* vec) const;

1 Returns: do_is(m, c) or do_is(low, high, vec).

const charT* scan_is(mask m, const charT* low, const charT* high) const;
2 Returns: do_scan_is(m, low, high).

const charT* scan_not(mask m, const charT* low, const charT* high) const;
3 Returns: do_scan_not(m, low, high).

charT    toupper(charT) const;
const charT* toupper(const charT* low, const charT* high) const;
4 Returns: do_toupper(c) or do_toupper(low, high).

charT    tolower(charT) const;
const charT* tolower(const charT* low, const charT* high) const;
5 Returns: do_tolower(c) or do_tolower(low, high).

charT    widen(char c) const;
const charT* widen(const charT* low, const charT* high, charT* to) const;
6 Returns: do_widen(c) or do_widen(low, high, to).

char    narrow(charT c, char dfault) const;
const charT* narrow(const charT* low, const charT* high, char dfault, char* to) const;
7 Returns: do_narrow(c, dfault) or do_narrow(low, high, dfault, to).

28.4.2.2.3 ctype virtual functions

bool     do_is(mask m, charT c) const;
const charT* do_is(const charT* low, const charT* high, mask* vec) const;
1 Effects: Classifies a character or sequence of characters. For each argument character, identifies a value M of type ctype_base::mask. The second form identifies a value M of type ctype_base::mask for each *p where (low <= p && p < high), and places it into vec[p - low].

2 Returns: The first form returns the result of the expression (M & m) != 0; i.e., true if the character has the characteristics specified. The second form returns high.

const charT* do_scan_is(mask m, const charT* low, const charT* high) const;
3 Effects: Locates a character in a buffer that conforms to a classification m.

4 Returns: The smallest pointer p in the range [low, high) such that is(m, *p) would return true; otherwise, returns high.

const charT* do_scan_not(mask m, const charT* low, const charT* high) const;
5 Effects: Locates a character in a buffer that fails to conform to a classification m.

6 Returns: The smallest pointer p, if any, in the range [low, high) such that is(m, *p) would return false; otherwise, returns high.
charT do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;

Effects: Converts a character or characters to upper case. The second form replaces each character *p in the range [low, high] for which a corresponding upper-case character exists, with that character.

Returns: The first form returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form returns high.

charT dotolower(charT c) const;
const charT* dotolower(charT* low, const charT* high) const;

Effects: Converts a character or characters to lower case. The second form replaces each character *p in the range [low, high] and for which a corresponding lower-case character exists, with that character.

Returns: The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns high.

charT do_widen(char c) const;
const char* do_widen(const char* low, const char* high, charT* dest) const;

Effects: Applies the simplest reasonable transformation from a char value or sequence of char values to the corresponding charT value or values. The only characters for which unique transformations are required are those in the basic source character set (5.3).

For any named ctype category with a ctype <charT> facet ctc and valid ctype_base::mask value M, (ctc.is(M, c) || !is(M, do_widen(c)) ) is true.

The second form transforms each character *p in the range [low, high], placing the result in dest[p - low].

Returns: The first form returns the transformed value. The second form returns high.

char do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high, char dfault, char* dest) const;

Effects: Applies the simplest reasonable transformation from a charT value or sequence of charT values to the corresponding char value or values. For any character c in the basic source character set (5.3) the transformation is such that do_widen(do_narrow(c, 0)) == c

For any named ctype category with a ctype<char> facet ctc however, and ctype_base::mask value M,

(is(M, c) || !ctc.is(M, do_narrow(c, dfault)) )

is true (unless do_narrow returns dfault). In addition, for any digit character c, the expression (do_narrow(c, dfault) - '0') evaluates to the digit value of the character. The second form transforms each character *p in the range [low, high], placing the result (or dfault if no simple transformation is readily available) in dest[p - low].

Returns: The first form returns the transformed value; or dfault if no mapping is readily available. The second form returns high.

28.4.2.3 Class template ctype_byname

namespace std {
  template<class charT>
  class ctype_byname : public ctype<charT> {
  public:
    using mask = typename ctype<charT>::mask;
    explicit ctype_byname(const char*, size_t refs = 0);
    explicit ctype_byname(const string&, size_t refs = 0);

  protected:
    ~ctype_byname();

 266) The char argument of do_widen is intended to accept values derived from character-literals for conversion to the locale's encoding.
267) In other words, the transformed character is not a member of any character classification that c is not also a member of.
28.4.2.4.1 General

```cpp
namespace std {
    template<>
    class ctype<char> : public locale::facet, public ctype_base {
        using char_type = char;
        explicit ctype(const mask* tab = nullptr, bool del = false, size_t refs = 0);
        bool is(mask m, char c) const;
        const char* is(const char* low, const char* high, mask* vec) const;
        const char* scan_is (mask m, const char* low, const char* high) const;
        const char* scan_not(mask m, const char* low, const char* high) const;
        char toupper(char c) const;
        const char* toupper(char* low, const char* high) const;
        char tolower(char c) const;
        const char* tolower(char* low, const char* high) const;
        char widen(char c) const;
        const char* widen(const char* low, const char* high, char* to) const;
        char narrow(char c, char dfault) const;
        const char* narrow(const char* low, const char* high, char dfault, char* to) const;
    };
}
```

A specialization `ctype<char>` is provided so that the member functions on type `char` can be implemented inline.\(^\text{268}\) The implementation-defined value of member `table_size` is at least 256.

28.4.2.4.2 Destructor

```
~ctype();
```

Effects: If the constructor’s first argument was nonzero, and its second argument was `true`, does `delete [] table();`.

---

\(^\text{268}\) Only the `char` (not `unsigned char` and `signed char`) form is provided. The specialization is specified in the standard, and not left as an implementation detail, because it affects the derivation interface for `ctype<char>`.

§ 28.4.2.4.2
28.4.2.4.3 Members

In the following member descriptions, for unsigned char values \( v \) where \( v \geq \text{table_size} \), \( \text{table()}[v] \) is assumed to have an implementation-specific value (possibly different for each such value \( v \)) without performing the array lookup.

```cpp
explicit ctype(const mask* tbl = nullptr, bool del = false, size_t refs = 0);
```

**Preconditions:** Either \( tbl == \text{nullptr} \) is true or \([tbl, tbl + \text{table_size})\) is a valid range.

**Effects:** Passes its `refs` argument to its base class constructor.

```cpp
bool is(mask m, char c) const;
const char* is(const char* low, const char* high, mask* vec) const;
```

**Effects:** The second form, for all \(*p\) in the range \([low, high)\), assigns into \( \text{vec}[p - \text{low}] \) the value \( \text{table()}[(\text{unsigned char})*p] \).

**Returns:** The first form returns \( \text{table()}[(\text{unsigned char})c] \& m \); the second form returns `high`.

```cpp
const char* scan_is(mask m, const char* low, const char* high) const;
```

**Returns:** The smallest \( p \) in the range \([low, high)\) such that \( \text{table()}[(\text{unsigned char}) *p] \& m \) is true.

```cpp
const char* scan_not(mask m, const char* low, const char* high) const;
```

**Returns:** The smallest \( p \) in the range \([low, high)\) such that \( \text{table()}[(\text{unsigned char}) *p] \& m \) is false.

```cpp
char toupper(char c) const;
const char* toupper(char* low, const char* high) const;
```

**Returns:** `do_toupper(c)` or `do_toupper(low, high)`, respectively.

```cpp
char tolower(char c) const;
const char* tolower(char* low, const char* high) const;
```

**Returns:** `do_tolower(c)` or `do_tolower(low, high)`, respectively.

```cpp
char widen(char c) const;
const char* widen(const char* low, const char* high, char* to) const;
```

**Returns:** `do_widen(c)` or `do_widen(low, high, to)`, respectively.

```cpp
char narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high, char* to) const;
```

**Returns:** `do_narrow(c, dfault)` or `do_narrow(low, high, dfault, to)`, respectively.

```cpp
const mask* table() const noexcept;
```

**Returns:** The first constructor argument, if it was nonzero, otherwise `classic_table()`.

28.4.2.4.4 Static members

```cpp
static const mask* classic_table() noexcept;
```

**Returns:** A pointer to the initial element of an array of size `table_size` which represents the classifications of characters in the "C" locale.

28.4.2.4.5 Virtual functions

```cpp
char do_toupper(char) const;
const char* do_toupper(char* low, const char* high) const;
char do_tolower(char) const;
const char* do_tolower(char* low, const char* high) const;
```
virtual char do_widen(char c) const;
virtual const char* do_widen(const char* low, const char* high, char* to) const;
virtual char do_narrow(char c, char dfault) const;
virtual const char* do_narrow(const char* low, const char* high, char dfault, char* to) const;

1 These functions are described identically as those members of the same name in the ctype class template (28.4.2.2.2).

28.4.2.5 Class template codecvt

28.4.2.5.1 General

namespace std {
    class codecvt_base {
        public:
            enum result { ok, partial, error, noconv }; 
    }

template<class internT, class externT, class stateT>
class codecvt : public locale::facet, public codecvt_base {
    public:
        using intern_type = internT;
        using extern_type = externT;
        using state_type = stateT;

        explicit codecvt(size_t refs = 0);

        result out(
            stateT& state,
            const internT* from, const internT* from_end, const internT*& from_next,
            externT* to, externT* to_end, externT*& to_next) const;

        result unshift(
            stateT& state,
            externT* to, externT* to_end, externT*& to_next) const;

        result in(
            stateT& state,
            const externT* from, const externT* from_end, const externT*& from_next,
            internT* to, internT* to_end, internT*& to_next) const;

        int encoding() const noexcept;
        bool always_noconv() const noexcept;
        int length(stateT& state, const externT* from, const externT* end, size_t max) const;
        int max_length() const noexcept;

        static locale::id id;
    }
}

protected:
    ~codecvt();
    virtual result do_out(
        stateT& state,
        const internT* from, const internT* from_end, const internT*& from_next,
        externT* to, externT* to_end, externT*& to_next) const;

    virtual result do_in(
        stateT& state,
        const externT* from, const externT* from_end, const externT*& from_next,
        internT* to, internT* to_end, internT*& to_next) const;

    virtual result do_unshift(
        stateT& state,
        externT* to, externT* to_end, externT*& to_next) const;

    virtual int do_encoding() const noexcept;
    virtual bool do_always_noconv() const noexcept;
    virtual int do_length(stateT& state, const externT* from, const externT* end, size_t max) const;

§ 28.4.2.5.1
virtual int do_max_length() const noexcept;
};
}

The class `codecvt<internT, externT, stateT>` is for use when converting from one character encoding to another, such as from wide characters to multibyte characters or between wide character encodings such as UTF-32 and EUC.

The `stateT` argument selects the pair of character encodings being mapped between.

The specializations required in Table 102 (28.3.1.2.1) convert the implementation-defined native character set. `codecvt<char, char, mbstate_t>` implements a degenerate conversion; it does not convert at all. The specialization `codecvt<char16_t, char8_t, mbstate_t>` converts between the UTF-16 and UTF-8 encoding forms, and the specialization `codecvt<char32_t, char8_t, mbstate_t>` converts between the UTF-32 and UTF-8 encoding forms. `codecvt<wchar_t, char, mbstate_t>` converts between the native character sets for ordinary and wide characters. Specializations on `mbstate_t` perform conversion between encodings known to the library implementer. Other encodings can be converted by specializing on a programmer-defined `stateT` type. Objects of type `stateT` can contain any state that is useful to communicate to or from the specialized `do_in` or `do_out` members.

### 28.4.2.5.2 Members

#### [locale.codecvt.members]

```cpp
result out(
    stateT& state,
    const internT* from, const internT* from_end, const internT*& from_next,
    externT* to, externT* to_end, externT*& to_next) const;
1 Returns: do_out(state, from, from_end, from_next, to, to_end, to_next).
result unshift(stateT& state, externT* to, externT* to_end, externT*& to_next) const;
2 Returns: do_unshift(state, to, to_end, to_next).
result in(
    stateT& state,
    const externT* from, const externT* from_end, const externT*& from_next,
    internT* to, internT* to_end, internT*& to_next) const;
3 Returns: do_in(state, from, from_end, from_next, to, to_end, to_next).
int encoding() const noexcept;
4 Returns: do_encoding().
bool always_noconv() const noexcept;
5 Returns: do_always_noconv().
int length(stateT& state, const externT* from, const externT* from_end, size_t max) const;
6 Returns: do_length(state, from, from_end, max).
int max_length() const noexcept;
7 Returns: do_max_length().
```

### 28.4.2.5.3 Virtual functions

#### [locale.codecvt.virtuals]

```cpp
result do_out(
    stateT& state,
    const internT* from, const internT* from_end, const internT*& from_next,
    externT* to, externT* to_end, externT*& to_next) const;
1 Preconditions: (from <= from_end && to <= to_end) is well-defined and true; state is initialized, if at the beginning of a sequence, or else is equal to the result of converting the preceding characters in the sequence.
result do_in(
    stateT& state,
    const externT* from, const externT* from_end, const externT*& from_next,
    internT* to, internT* to_end, internT*& to_next) const;
```
Effects: Translates characters in the source range \([\text{from}, \text{from}_\text{end})\), placing the results in sequential positions starting at destination \(\text{to}\). Converts no more than \((\text{from}_\text{end} - \text{from})\) source elements, and stores no more than \((\text{to}_\text{end} - \text{to})\) destination elements.

3

Stops if it encounters a character it cannot convert. It always leaves the \(\text{from}_\text{next}\) and \(\text{to}_\text{next}\) pointers pointing one beyond the last element successfully converted. If returns \(\text{noconv}\), \(\text{internT}\) and \(\text{externT}\) are the same type and the converted sequence is identical to the input sequence \([\text{from}, \text{from}_\text{next})\). \(\text{to}_\text{next}\) is set equal to \(\text{to}\), the value of \(\text{state}\) is unchanged, and there are no changes to the values in \([\text{to}, \text{to}_\text{end})\).

4

A \(\text{codecvt}\) facet that is used by \(\text{basic_filebuf}\) \((29.9)\) shall have the property that if

\[
\text{do_out}(\text{state}, \text{from}, \text{from}_\text{end}, \text{from}_\text{next}, \text{to}, \text{to}_\text{end}, \text{to}_\text{next})
\]

would return \(\text{ok}\), where \(\text{from} \neq \text{from}_\text{end}\), then

\[
\text{do_out}(\text{state}, \text{from}, \text{from} + 1, \text{from}_\text{next}, \text{to}, \text{to}_\text{end}, \text{to}_\text{next})
\]

shall also return \(\text{ok}\), and that if

\[
\text{do_in}(\text{state}, \text{from}, \text{from}_\text{end}, \text{from}_\text{next}, \text{to}, \text{to}_\text{end}, \text{to}_\text{next})
\]

would return \(\text{ok}\), where \(\text{to} \neq \text{to}_\text{end}\), then

\[
\text{do_in}(\text{state}, \text{from}, \text{from}_\text{end}, \text{from}_\text{next}, \text{to}, \text{to} + 1, \text{to}_\text{next})
\]

shall also return \(\text{ok}\).

\[\text{Note 1}: \text{As a result of operations on} \ \text{state}, \text{it can return} \ \text{ok} \text{ or} \partial \text{tal and set from}_\text{next} = \text{from} \text{ and to}_\text{next} = \text{to}. \text{—end note}\]

Remarks: Its operations on \(\text{state}\) are unspecified.

\[\text{Note 2}: \text{This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets.} \text{—end note}\]

Returns: An enumeration value, as summarized in Table 104.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>completed the conversion</td>
</tr>
<tr>
<td>partial</td>
<td>not all source characters converted</td>
</tr>
<tr>
<td>error</td>
<td>encountered a character in ([\text{from}, \text{from}_\text{end})) that it could not convert</td>
</tr>
<tr>
<td>noconv</td>
<td>(\text{internT}) and (\text{externT}) are the same type, and input sequence is identical to converted sequence</td>
</tr>
</tbody>
</table>

A return value of \(\partial \text{tal}, \text{if} \ (\text{from}_\text{next} = \text{from}_\text{end})\), indicates that either the destination sequence has not absorbed all the available destination elements, or that additional source elements are needed before another destination element can be produced.

result \(\text{do_unshift(stateT& state, externT* to, externT* to}_\text{end}, \text{externT}* to}_\text{next})\) const;

Preconditions: \((\text{to} \leq \text{to}_\text{end})\) is well-defined and \(\text{true}; \text{state}\) is initialized, if at the beginning of a sequence, or else is equal to the result of converting the preceding characters in the sequence.

Effects: Places characters starting at \(\text{to}\) that should be appended to terminate a sequence when the current \(\text{stateT}\) is given by \(\text{state}\)\(^{270}\). Stores no more than \((\text{to}_\text{end} - \text{to})\) destination elements, and leaves the \(\text{to}_\text{next}\) pointer pointing one beyond the last element successfully stored.

Returns: An enumeration value, as summarized in Table 105.

---

\(^{269}\) Informally, this means that \(\text{basic_filebuf}\) assumes that the mappings from internal to external characters is 1 to \(N\): that a \(\text{codecvt}\) facet that is used by \(\text{basic_filebuf}\) can translate characters one internal character at a time.

\(^{270}\) Typically these will be characters to return the state to \(\text{stateT}()\).
Table 105: do_unshift result values  [tab:locale.codecvt.unshift]

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>completed the sequence</td>
</tr>
<tr>
<td>partial</td>
<td>space for more than to_end - to destination elements was needed to terminate a sequence given the value of state</td>
</tr>
<tr>
<td>error</td>
<td>an unspecified error has occurred</td>
</tr>
<tr>
<td>noconv</td>
<td>no termination is needed for this state_type</td>
</tr>
</tbody>
</table>

int do_encoding() const noexcept;

Returns: -1 if the encoding of the externT sequence is state-dependent; else the constant number of externT characters needed to produce an internal character; or 0 if this number is not a constant.\textsuperscript{271}

bool do_always_noconv() const noexcept;

Returns: true if do_in() and do_out() return noconv for all valid argument values. codecvt<char, char, mbstate_t> returns true.

int do_length(stateT& state, const externT* from, const externT* from_end, size_t max) const;

Preconditions: (from <= from_end) is well-defined and true; state is initialized, if at the beginning of a sequence, or else is equal to the result of converting the preceding characters in the sequence.

Effects: The effect on the state argument is as if it called do_in(state, from, from_end, from, to, to+max, to) for to pointing to a buffer of at least max elements.

Returns: (from_next-from) where from_next is the largest value in the range [from, from_end] such that the sequence of values in the range [from, from_next) represents max or fewer valid complete characters of type internT. The specialization codecvt<char, char, mbstate_t> returns the lesser of max and (from_end-from).

int do_max_length() const noexcept;

Returns: The maximum value that do_length(state, from, from_end, 1) can return for any valid range [from, from_end] and stateT value state. The specialization codecvt<char, char, mbstate_t>::do_max_length() returns 1.

28.4.2.6 Class template codecvtbyname  [locale.codecvtbyname]

namespace std {
  template<class internT, class externT, class stateT>
  class codecvtbyname : public codecvt<internT, externT, stateT> {
    public:
      explicit codecvtbyname(const char*, size_t refs = 0);
      explicit codecvtbyname(const string&, size_t refs = 0);
      ...
  };
}

28.4.3 The numeric category  [category.numeric]

28.4.3.1 General  [category.numeric.general]

The classes num_get<> and num_put<> handle numeric formatting and parsing. Virtual functions are provided for several numeric types. Implementations may (but are not required to) delegate extraction of smaller types to extractors for larger types.\textsuperscript{272}

\textsuperscript{271} If encoding() yields -1, then more than max_length() externT elements can be consumed when producing a single internT character, and additional externT elements can appear at the end of a sequence after those that yield the final internT character.

\textsuperscript{272} Parsing "-1" correctly into, e.g., an unsigned short requires that the corresponding member get() at least extract the sign before delegating.
All specifications of member functions for `num_put` and `num_get` in the subclauses of 28.4.3 only apply to the specializations required in Tables 102 and 103 (28.3.1.2.1), namely `num_get<char>`, `num_get<wchar_t>`, `num_get<InputIterator>`, `num_put<char>`, `num_put<wchar_t>`, and `num_put<InputIterator>`. These specializations refer to the `ios_base`\& argument for formatting specifications (28.4), and to its imbued locale for the `numpunct<locale>` facet to identify all numeric punctuation preferences, and also for the `ctype<locale>` facet to perform character classification.

Extractor and inserter members of the standard iostreams use `num_get<>` and `num_put<>` member functions for formatting and parsing numeric values (29.7.4.3.1, 29.7.5.3.1).

28.4.3.2 Class template `num_get`  
28.4.3.2.1 General

```cpp
namespace std {
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class num_get : public locale::facet {
    public:
        using char_type = charT;
        using iter_type = InputIterator;
        explicit num_get(size_t refs = 0);

        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, bool& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, long& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, long long& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, unsigned short& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, unsigned int& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, unsigned long& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, unsigned long long& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, float& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, double& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, long double& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&
                      , ios_base::iostate& err, void*& v) const;

        static locale::id id;
    protected:
        ~num_get();
        virtual iter_type do_get(iter_type in, iter_type end, ios_base&
                                 , ios_base::iostate& err, bool& v) const;
        virtual iter_type do_get(iter_type in, iter_type end, ios_base&
                                 , ios_base::iostate& err, long& v) const;
        virtual iter_type do_get(iter_type in, iter_type end, ios_base&
                                 , ios_base::iostate& err, long long& v) const;
        virtual iter_type do_get(iter_type in, iter_type end, ios_base&
                                 , ios_base::iostate& err, unsigned short& v) const;
        virtual iter_type do_get(iter_type in, iter_type end, ios_base&
                                 , ios_base::iostate& err, unsigned int& v) const;
        virtual iter_type do_get(iter_type in, iter_type end, ios_base&
                                 , ios_base::iostate& err, unsigned long& v) const;
        virtual iter_type do_get(iter_type in, iter_type end, ios_base&
                                 , ios_base::iostate& err, unsigned long long& v) const;
    }
```
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, float& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, double& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, long double& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, void*& v) const;
};
}

The facet num_get is used to parse numeric values from an input sequence such as an istream.

28.4.3.2.2 Members [facet.num.get.members]

iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, bool& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned short& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned int& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, float& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, void*& val) const;

1 Returns: do_get(in, end, str, err, val).

28.4.3.2.3 Virtual functions [facet.num.get.virtuals]

iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned short& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned int& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, float& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, void*& val) const;

1 Effects: Reads characters from in, interpreting them according to str.flags(), use_facet<ctype<
    charT>>(loc), and use_facet<numpunct<charT>>(loc), where loc is str.getloc().

The details of this operation occur in three stages

§ 28.4.3.2.3
Stage 1: Determine a conversion specifier

Stage 2: Extract characters from \texttt{in} and determine a corresponding char value for the format expected by the conversion specification determined in stage 1.

Stage 3: Store results

The details of the stages are presented below.

**Stage 1:** The function initializes local variables via

\begin{verbatim}
fmtflags flags = str.flags();
fmtflags basefield = (flags & ios_base::basefield);
fmtflags uppercase = (flags & ios_base::uppercase);
fmtflags boolalpha = (flags & ios_base::boolalpha);
\end{verbatim}

For conversion to an integral type, the function determines the integral conversion specifier as indicated in Table 106. The table is ordered. That is, the first line whose condition is true applies.

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basefield == oct</td>
<td>\texttt{%o}</td>
</tr>
<tr>
<td>basefield == hex</td>
<td>\texttt{%X}</td>
</tr>
<tr>
<td>basefield == 0</td>
<td>\texttt{%i}</td>
</tr>
<tr>
<td>signed integral type</td>
<td>\texttt{%d}</td>
</tr>
<tr>
<td>unsigned integral type</td>
<td>\texttt{%u}</td>
</tr>
</tbody>
</table>

For conversions to a floating-point type the specifier is \texttt{%g}.

For conversions to \texttt{void*} the specifier is \texttt{%p}.

A length modifier is added to the conversion specification, if needed, as indicated in Table 107.

<table>
<thead>
<tr>
<th>Type</th>
<th>Length modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>\texttt{h}</td>
</tr>
<tr>
<td>unsigned short</td>
<td>\texttt{h}</td>
</tr>
<tr>
<td>long</td>
<td>\texttt{l}</td>
</tr>
<tr>
<td>unsigned long</td>
<td>\texttt{l}</td>
</tr>
<tr>
<td>long long</td>
<td>\texttt{ll}</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>\texttt{ll}</td>
</tr>
<tr>
<td>double</td>
<td>\texttt{l}</td>
</tr>
<tr>
<td>long double</td>
<td>\texttt{L}</td>
</tr>
</tbody>
</table>

**Stage 2:** If \texttt{in} == \texttt{end} then stage 2 terminates. Otherwise a charT is taken from \texttt{in} and local variables are initialized as if by

\begin{verbatim}
char_type ct = *in;
char c = src[find(atoms, atoms + sizeof(src) - 1, ct) - atoms];
if (ct == use_facet<numpunct<charT>>>(loc).decimal_point())
  c = '.';?
bool discard =
  ct == use_facet<numpunct<charT>>>(loc).thousands_sep()
  && use_facet<numpunct<charT>>>(loc).grouping().length() != 0;
\end{verbatim}

where the values src and atoms are defined as if by:

\begin{verbatim}
static const char src[] = "0123456789abcdefghABCDEFX++";
char_type atoms[sizeof(src)];
use_facet<ctype<charT>>>(loc).widen(src, src + sizeof(src), atoms);
\end{verbatim}

for this value of \texttt{loc}.

If \texttt{discard} is \texttt{true}, then if "." has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if "." has already been accumulated, the character is discarded and Stage 2 terminates. If it is not discarded, then a

\textsection{28.4.3.2.3 1354}
check is made to determine if \( c \) is allowed as the next character of an input field of the conversion specifier returned by Stage 1. If so, it is accumulated.

If the character is either discarded or accumulated then \( in \) is advanced by \( ++in \) and processing returns to the beginning of stage 2.

**Stage 3:** The sequence of \( \texttt{chars} \) accumulated in stage 2 (the field) is converted to a numeric value by the rules of one of the functions declared in the header \(<\texttt{cstdlib}>>\):

- For a signed integer value, the function \texttt{strtol}.
- For an unsigned integer value, the function \texttt{strtoull}.
- For a \texttt{float} value, the function \texttt{strtod}.
- For a \texttt{double} value, the function \texttt{strtold}.
- For a \texttt{long double} value, the function \texttt{strtold}.

The numeric value to be stored can be one of:

- zero, if the conversion function does not convert the entire field.
- the most positive (or negative) representable value, if the field to be converted to a signed integer type represents a value too large positive (or negative) to be represented in \( val \).
- the most positive representable value, if the field to be converted to an unsigned integer type represents a value that cannot be represented in \( val \).
- the converted value, otherwise.

The resultant numeric value is stored in \( val \). If the conversion function does not convert the entire field, or if the field represents a value outside the range of representable values, \texttt{ios\_base::failbit} is assigned to \texttt{err}.

Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with \texttt{use\_facet\<\texttt{numpunct\<charT\>\>(loc).grouping()}}. If they are not consistent then \texttt{ios\_base::failbit} is assigned to \texttt{err}.

In any case, if stage 2 processing was terminated by the test for \( in == \) \texttt{end} then \texttt{err \|= ios\_base::eofbit} is performed.

```cpp
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, bool& val) const;
```

**Effects:** If \((\texttt{str.flags()\&ios\_base::boolalpha}) == 0\) then input proceeds as it would for a \texttt{long} except that if a value is being stored into \( val \), the value is determined according to the following: If the value to be stored is 0 then \texttt{false} is stored. If the value is 1 then \texttt{true} is stored. Otherwise \texttt{true} is stored and \texttt{ios\_base::failbit} is assigned to \texttt{err}.

Otherwise target sequences are determined “as if” by calling the members \texttt{false\_name()} and \texttt{true\_name()} of the facet obtained by \texttt{use\_facet\<\texttt{numpunct\<charT\>\>(str.getloc())}}. Successive characters in the range \([in, end]) \( \texttt{(see} \texttt{22.2.3)} \) are obtained and matched against corresponding positions in the target sequences only as necessary to identify a unique match. The input iterator \( in \) is compared to \texttt{end} only when necessary to obtain a character. If a target sequence is uniquely matched, \( val \) is set to the corresponding value. Otherwise \texttt{false} is stored and \texttt{ios\_base::failbit} is assigned to \texttt{err}.

The in iterator is always left pointing one position beyond the last character successfully matched. If \( val \) is set, then \texttt{err} is set to \texttt{str\_goodbit}; or to \texttt{str\_eofbit} if, when seeking another character to match, it is found that \((in == \texttt{end})\). If \( val \) is not set, then \texttt{err} is set to \texttt{str\_failbit}; or to \texttt{(str\_failbit|str\_eofbit)} if the reason for the failure was that \((in == \texttt{end})\).

[Example 1: For targets true: "a" and false: "abb", the input sequence "a" yields \( val == \texttt{true} \) and \texttt{err == str\_eofbit}; the input sequence "abc" yields \( err == \texttt{str\_failbit}, \) with \texttt{in} ending at the ‘\texttt{c}’ element. For targets true: "1" and false: "0", the input sequence "1" yields \( val == \texttt{true} \) and \texttt{err == str\_goodbit}. For empty targets (""), any input sequence yields \( err == \texttt{str\_failbit}. \) — end example]

**Returns:** \texttt{in}.

### 28.4.3.3 Class template num_put

#### 28.4.3.3.1 General

```cpp
namespace std {
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class num_put : public locale::facet {
        public:
            using char_type = charT;

```
using iter_type = OutputIterator;

explicit num_put(size_t refs = 0);

iter_type put(iter_type s, ios_base& f, char_type fill, bool v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, long v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, long long v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, unsigned long v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, unsigned long long v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, double v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, long double v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, const void* v) const;

static locale::id id;

protected:
- num_put();
virtual iter_type do_put(iter_type, ios_base&, char_type fill, bool v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, long v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, long long v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, unsigned long v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, unsigned long long v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, double v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, long double v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, const void* v) const;
}

The facet **num_put** is used to format numeric values to a character sequence such as an ostream.

### 28.4.3.3.2 Members

[facet.num.put.members]

iter_type put(iter_type out, ios_base& str, char_type fill, bool val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, unsigned long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, unsigned long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, long double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, const void* val) const;

1 Returns: do_put(out, str, fill, val).

### 28.4.3.3 Virtual functions

[facet.num.put.virtuals]

iter_type do_put(iter_type out, ios_base& str, char_type fill, long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, long long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, double val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, long double val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, const void* val) const;

1 Effects: Writes characters to the sequence out, formatting val as desired. In the following description, loc names a local variable initialized as

locale loc = str.getloc();

2 The details of this operation occur in several stages:

(2.1) Stage 1: Determine a printf conversion specifier \texttt{spec} and determine the characters that would be printed by \texttt{printf} (29.12) given this conversion specifier for

\texttt{printf(spec, val)}

assuming that the current locale is the "C" locale.

(2.2) Stage 2: Adjust the representation by converting each \texttt{char} determined by stage 1 to a \texttt{charT} using a conversion and values returned by members of use_facet\texttt{<numpunct<charT>>}(loc).

\section*{§ 28.4.3.3.3}

1356
Stage 3: Determine where padding is required.

Stage 4: Insert the sequence into the `out`.

Detailed descriptions of each stage follow.

**Returns:** `out`.

**Stage 1:** The first action of stage 1 is to determine a conversion specifier. The tables that describe this determination use the following local variables:

- `fmtflags flags = str.flags();`
- `fmtflags basefield = (flags & (ios_base::basefield));`
- `fmtflags uppercase = (flags & (ios_base::uppercase));`
- `fmtflags floatfield = (flags & (ios_base::floatfield));`
- `fmtflags showpos = (flags & (ios_base::showpos));`
- `fmtflags showbase = (flags & (ios_base::showbase));`
- `fmtflags showpoint = (flags & (ios_base::showpoint));`

All tables used in describing stage 1 are ordered. That is, the first line whose condition is true applies. A line without a condition is the default behavior when none of the earlier lines apply.

For conversion from an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 108.

**Table 108: Integer conversions**

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>basefield == ios_base::oct</code></td>
<td><code>%o</code></td>
</tr>
<tr>
<td><code>(basefield == ios_base::hex) &amp;&amp; !uppercase</code></td>
<td><code>%x</code></td>
</tr>
<tr>
<td><code>(basefield == ios_base::hex)</code></td>
<td><code>%X</code></td>
</tr>
<tr>
<td>for a <strong>signed</strong> integral type</td>
<td><code>%d</code></td>
</tr>
<tr>
<td>for an <strong>unsigned</strong> integral type</td>
<td><code>%u</code></td>
</tr>
</tbody>
</table>

For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 109.

**Table 109: Floating-point conversions**

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>floatfield == ios_base::fixed</code></td>
<td><code>%f</code></td>
</tr>
<tr>
<td><code>floatfield == ios_base::scientific &amp;&amp; !uppercase</code></td>
<td><code>%e</code></td>
</tr>
<tr>
<td><code>floatfield == ios_base::scientific</code></td>
<td><code>%E</code></td>
</tr>
<tr>
<td>`floatfield == (ios_base::fixed</td>
<td>ios_base::scientific) &amp;&amp; !uppercase`</td>
</tr>
<tr>
<td>`floatfield == (ios_base::fixed</td>
<td>ios_base::scientific)`</td>
</tr>
<tr>
<td><code>!uppercase</code></td>
<td><code>%g</code></td>
</tr>
<tr>
<td><strong>otherwise</strong></td>
<td><code>%G</code></td>
</tr>
</tbody>
</table>

For conversions from an integral or floating-point type a length modifier is added to the conversion specifier as indicated in Table 110.

**Table 110: Length modifier**

<table>
<thead>
<tr>
<th>Type</th>
<th>Length modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>1</td>
</tr>
<tr>
<td>long long</td>
<td>1l</td>
</tr>
<tr>
<td>unsigned long</td>
<td>1</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>1l</td>
</tr>
<tr>
<td>long double</td>
<td>L</td>
</tr>
<tr>
<td><strong>otherwise</strong></td>
<td><code>none</code></td>
</tr>
</tbody>
</table>

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 111.
Table 111: Numeric conversions  

<table>
<thead>
<tr>
<th>Type(s)</th>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>an integral type</td>
<td>showpos +</td>
<td></td>
</tr>
<tr>
<td></td>
<td>showbase #</td>
<td></td>
</tr>
<tr>
<td>a floating-point type</td>
<td>showpos +</td>
<td></td>
</tr>
<tr>
<td></td>
<td>showpoint #</td>
<td></td>
</tr>
</tbody>
</table>

For conversion from a floating-point type, if `floatfield != (ios_base::fixed | ios_base::scientific), str.precision()` is specified as precision in the conversion specification. Otherwise, no precision is specified.

For conversion from `void*` the specifier is `%p`.

The representations at the end of stage 1 consists of the `char's that would be printed by a call of `printf(s, val)` where `s` is the conversion specifier determined above.

**Stage 2:** Any character `c` other than a decimal point(.) is converted to a `charT` via

```cpp
use_facet<ctype<charT>>(loc).widen(c)
```

A local variable `punct` is initialized via

```cpp
const numpunct<charT>& punct = use_facet<numpunct<charT>>(loc);
```

For arithmetic types, `punct.thousands_sep()` characters are inserted into the sequence as determined by the value returned by `punct.do_grouping()` using the method described in 28.4.4.1.3 Decimal point characters(.) are replaced by `punct.decimal_point()`

**Stage 3:** A local variable is initialized as

```cpp
fmtflags adjustfield = (flags & (ios_base::adjustfield));
```

The location of any padding is determined according to **Table 112.**

<table>
<thead>
<tr>
<th>State</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustfield == ios_base::left</td>
<td>pad after</td>
</tr>
<tr>
<td>adjustfield == ios_base::right</td>
<td>pad before</td>
</tr>
<tr>
<td>adjustfield == internal and a sign occurs in the representation</td>
<td>pad after the sign</td>
</tr>
<tr>
<td>adjustfield == internal and representation after stage 1 began with 0x or 0X</td>
<td>pad after x or X</td>
</tr>
<tr>
<td>otherwise</td>
<td>pad before</td>
</tr>
</tbody>
</table>

If `str.width()` is nonzero and the number of `charT's in the sequence after stage 2 is less than `str.width()`, then enough fill characters are added to the sequence at the position indicated for padding to bring the length of the sequence to `str.width()`.

`str.width(0)` is called.

**Stage 4:** The sequence of `charT's at the end of stage 3 are output via

```cpp
*out++ = c
```

```cpp
iter_type do_put(iter_type out, ios_base& str, char_type fill, bool val) const;
```

5  

Returns: If `(str.flags() & ios_base::boolalpha) == 0 returns do_put(out, str, fill, (int)val), otherwise obtains a string `s` as if by

```cpp
string_type s =
val ? use_facet<numpunct<charT>>(loc).truename()
: use_facet<numpunct<charT>>(loc).falsename();
```

and then inserts each character `c` of `s into out via *out++ = c and returns out.

---

273) The conversion specification `#o` generates a leading 0 which is not a padding character.
28.4.4 The numeric punctuation facet

28.4.4.1 Class template numpunct

28.4.4.1.1 General

namespace std {
    template<class charT>
    class numpunct : public locale::facet {
        using char_type = charT;
        using string_type = basic_string<charT>;

        explicit numpunct(size_t refs = 0);
        char_type decimal_point() const;
        char_type thousands_sep() const;
        string grouping() const;
        string_type truename() const;
        string_type falsename() const;
        static locale::id id;
    protected:
        -numpunct(); // virtual
        virtual char_type do_decimal_point() const;
        virtual char_type do_thousands_sep() const;
        virtual string do_grouping() const;
        virtual string_type do_truename() const; // for bool
        virtual string_type do_falsename() const; // for bool
    };
}

numpunct<> specifies numeric punctuation. The specializations required in Table 102 (28.3.1.2.1), namely numpunct<wchar_t> and numpunct<char>, provide classic "C" numeric formats, i.e., they contain information equivalent to that contained in the "C" locale or their wide character counterparts as if obtained by a call to widen.

The syntax for number formats is as follows, where digit represents the radix set specified by the fmtflags argument value, and thousands-sep and decimal-point are the results of corresponding numpunct<charT> members. Integer values have the format:

```
intval:
signopt units
sign:
+
-
units:
digits
digits thousands-sep units
digits:
digit digitsopt
```

and floating-point values have:

```
floatval:
signopt fractionalopt exponentopt
signopt decimal-point digits exponentopt
fractional:
decimal-point digitsopt
exponent:
e signopt digits
e:
e E
```
where the number of digits between thousands-seps is as specified by do_grouping(). For parsing, if the digits portion contains no thousands-separators, no grouping constraint is applied.

### 28.4.4.1.2 Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char_type decimal_point() const;</td>
<td>Returns: do_decimal_point().</td>
</tr>
<tr>
<td>char_type thousands_sep() const;</td>
<td>Returns: do_thousands_sep().</td>
</tr>
<tr>
<td>string grouping() const;</td>
<td>Returns: do_grouping().</td>
</tr>
<tr>
<td>string_type truename() const;</td>
<td>Returns: do_truename() or do_falsename(), respectively.</td>
</tr>
<tr>
<td>string_type falsename() const;</td>
<td></td>
</tr>
</tbody>
</table>

### 28.4.4.1.3 Virtual functions

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char_type do_decimal_point() const;</td>
<td>Returns: A character for use as the decimal radix separator. The required specializations return &quot;.&quot; or L&quot;.&quot;.</td>
</tr>
<tr>
<td>char_type do_thousands_sep() const;</td>
<td>Returns: A character for use as the digit group separator. The required specializations return &quot;,&quot; or L&quot;,&quot;.</td>
</tr>
<tr>
<td>string do_grouping() const;</td>
<td>Returns: A string vec used as a vector of integer values, in which each element vec[i] represents the number of digits in the group at position i, starting with position 0 as the rightmost group. If vec.size() &lt;= i, the number is the same as group (i - 1); if (i &lt; 0</td>
</tr>
<tr>
<td>string_type do_truename() const;</td>
<td>Returns: A string representing the name of the boolean value true or false, respectively.</td>
</tr>
<tr>
<td>string_type do_falsename() const;</td>
<td></td>
</tr>
</tbody>
</table>

### 28.4.4.2 Class template numpunctbyname

```cpp
namespace std {
    template<class charT>
    class numpunctbyname : public numpunct<charT> {
        // this class is specialized for char and wchar_t.
        public:
            using char_type = charT;
            using string_type = basic_string<charT>;

            explicit numpunctbyname(const char*, size_t refs = 0);
            explicit numpunctbyname(const string&, size_t refs = 0);

            protected:
                ~numpunctbyname();
        }
    }
}
```

---

274) Thus, the string "\003" specifies groups of 3 digits each, and "3" probably indicates groups of 51 (!) digits each, because 51 is the ASCII value of "3".
28.4.5 The collate category

28.4.5.1 Class template collate

28.4.5.1.1 General

namespace std {
    template<class charT>
    class collate : public locale::facet {
public:
    using char_type = charT;
    using string_type = basic_string<charT>;

    explicit collate(size_t refs = 0);
    int compare(const charT* low1, const charT* high1,
                const charT* low2, const charT* high2) const;
    string_type transform(const charT* low, const charT* high) const;
    long hash(const charT* low, const charT* high) const;

    static locale::id id;

protected:
    ~collate();
    virtual int do_compare(const charT* low1, const charT* high1,
                           const charT* low2, const charT* high2) const;
    virtual string_type do_transform(const charT* low, const charT* high) const;
    virtual long do_hash (const charT* low, const charT* high) const;
    
};

} /* std */

The class collate<charT> provides features for use in the collation (comparison) and hashing of strings. A locale member function template, operator(), uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (Clause 25) and containers operating on strings. The specializations required in Table 102 (28.3.1.2.1), namely collate<char> and collate<wchar_t>, apply lexicographic ordering (25.8.11).

Each function compares a string of characters *p in the range [low, high).

28.4.5.1.2 Members

int compare(const charT* low1, const charT* high1,
            const charT* low2, const charT* high2) const;
1
    Returns: do_compare(low1, high1, low2, high2).

string_type transform(const charT* low, const charT* high) const;
2
    Returns: do_transform(low, high).

long hash(const charT* low, const charT* high) const;
3
    Returns: do_hash(low, high).

28.4.5.1.3 Virtual functions

int do_compare(const charT* low1, const charT* high1,
               const charT* low2, const charT* high2) const;
1
    Returns: 1 if the first string is greater than the second, -1 if less, zero otherwise. The specializations required in Table 102 (28.3.1.2.1), namely collate<char> and collate<wchar_t>, implement a lexicographical comparison (25.8.11).

string_type do_transform(const charT* low, const charT* high) const;
2
    Returns: A basic_string<charT> value that, compared lexicographically with the result of calling transform() on another string, yields the same result as calling do_compare() on the same two strings.275

275) This function is useful when one string is being compared to many other strings.
long do_hash(const charT* low, const charT* high) const;

3 Returns: An integer value equal to the result of calling hash() on any other string for which do_compare() returns 0 (equal) when passed the two strings.

4 Recommended practice: The probability that the result equals that for another string which does not compare equal should be very small, approaching \(1.0/\text{numeric_limits<unsigned long>::max()}\).

28.4.5.2 Class template collate_byname

namespace std {
    template<class charT>
    class collate_byname : public collate<charT> {
    public:
        using string_type = basic_string<charT>;
        explicit collate_byname(const char*, size_t refs = 0);
        explicit collate_byname(const string&, size_t refs = 0);
        protected:
        -collate_byname();
    };
}

28.4.6 The time category

28.4.6.1 General

1 Templates time_get<charT, InputIterator> and time_put<charT, OutputIterator> provide date and time formatting and parsing. All specifications of member functions for time_put and time_get in the subclauses of 28.4.6 only apply to the specializations required in Tables 102 and 103 (28.3.1.2.1). Their members use their ios_base&, ios_base::iostate&, and fill arguments as described in 28.4, and the ctype<> facet, to determine formatting details.

28.4.6.2 Class template time_get

28.4.6.2.1 General

namespace std {
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class time_get : public locale::facet, public time_base {
    public:
        using char_type = charT;
        using iter_type = InputIterator;
        explicit time_get(size_t refs = 0);
        dateorder date_order() const { return do_date_order(); } const
        iter_type get_time(iter_type s, iter_type end, ios_base& f,
            ios_base::iostate& err, tm* t) const;
        iter_type get_date(iter_type s, iter_type end, ios_base& f,
            ios_base::iostate& err, tm* t) const;
        iter_type get_weekday(iter_type s, iter_type end, ios_base& f,
            ios_base::iostate& err, tm* t) const;
        iter_type get_monthname(iter_type s, iter_type end, ios_base& f,
            ios_base::iostate& err, tm* t) const;
        iter_type get_year(iter_type s, iter_type end, ios_base& f,
            ios_base::iostate& err, tm* t) const;
        iter_type get(iter_type s, iter_type end, ios_base& f,
            char format, char modifier = 0) const;
    };
}

§ 28.4.6.2.1 1362
static locale::id id;

protected:
- time_get();
  virtual dateorder do_date_order() const;
  virtual iter_type do_get_time(iter_type s, iter_type end, ios_base&,
      ios_base::iostate& err, tm* t) const;
  virtual iter_type do_get_date(iter_type s, iter_type end, ios_base&,
      ios_base::iostate& err, tm* t) const;
  virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base&,
      ios_base::iostate& err, tm* t) const;
  virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base&,
      ios_base::iostate& err, tm* t) const;
  virtual iter_type do_get_year(iter_type s, iter_type end, ios_base&,
      ios_base::iostate& err, tm* t) const;
  virtual iter_type do_get(iter_type s, iter_type end, ios_base& f,
      ios_base::iostate& err, tm* t, char format, char modifier) const;
};

1 time_get is used to parse a character sequence, extracting components of a time or date into a struct
   tm object. Each get member parses a format as produced by a corresponding format specifier to time-
   put<>::put. If the sequence being parsed matches the correct format, the corresponding members of the
   struct tm argument are set to the values used to produce the sequence; otherwise either an error is reported
   or unspecified values are assigned.\footnote{276}

2 If the end iterator is reached during parsing by any of the get() member functions, the member sets
   ios_base::eofbit in err.

28.4.6.2.2 Members [locale.time.get.members]

dateorder date_order() const;

   Returns: do_date_order().

   iter_type get_time(iter_type s, iter_type end, ios_base& str,
      ios_base::iostate& err, tm* t) const;

   Returns: do_get_time(s, end, str, err, t).

   iter_type get_date(iter_type s, iter_type end, ios_base& str,
      ios_base::iostate& err, tm* t) const;

   Returns: do_get_date(s, end, str, err, t).

   iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
      ios_base::iostate& err, tm* t) const;

   Returns: do_get_weekday(s, end, str, err, t) or do_get_monthname(s, end, str, err, t).

   iter_type get_monthname(iter_type s, iter_type end, ios_base& str,
      ios_base::iostate& err, tm* t) const;

   Returns: do_get_monthname(s, end, str, err, t).

   iter_type get_year(iter_type s, iter_type end, ios_base& str,
      ios_base::iostate& err, tm* t) const;

   Returns: do_get_year(s, end, str, err, t).

   iter_type get(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err,
      tm* t, char format, char modifier = 0) const;

   Returns: do_get(s, end, f, err, t, format, modifier).

   iter_type get(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err,
      tm* t, const char_type* fmt, const char_type* fmtend) const;

   Preconditions: [fmt, fmtend) is a valid range.

\footnote{276}{In other words, user confirmation is required for reliable parsing of user-entered dates and times, but machine-generated
  formats can be parsed reliably. This allows parsers to be aggressive about interpreting user variations on standard formats.}
Effects: The function starts by evaluating \( \text{err} = \text{ios\_base::goodbit} \). It then enters a loop, reading zero or more characters from \( s \) at each iteration. Unless otherwise specified below, the loop terminates when the first of the following conditions holds:

1. The expression \( \text{fmt} == \text{fmtend} \) evaluates to \( \text{true} \).
2. The expression \( \text{err} == \text{ios\_base::goodbit} \) evaluates to \( \text{false} \).
3. The expression \( s == \text{end} \) evaluates to \( \text{true} \), in which case the function evaluates \( \text{err} = \text{ios\_base::eofbit} \mid \text{ios\_base::failbit} \).
4. The next element of \( \text{fmt} \) is equal to ‘\%’, optionally followed by a modifier character, followed by a conversion specifier character, \( \text{format} \), together forming a conversion specification valid for the ISO/IEC 9945 function \( \text{strptime} \). If the number of elements in the range \( \text{[fmt, fmtend)} \) is not sufficient to unambiguously determine whether the conversion specification is complete and valid, the function evaluates \( \text{err} = \text{ios\_base::failbit} \). Otherwise, the function evaluates \( \text{s} = \text{do\_get(s, end, f, err, t, format, modifier)} \), where the value of \( \text{modifier} \) is ‘\0’ when the optional modifier is absent from the conversion specification. If \( \text{err} == \text{ios\_base::goodbit} \) holds after the evaluation of the expression, the function increments \( \text{fmt} \) to point just past the end of the conversion specification and continues looping.

5. The expression \( \text{isspace(*fmt, f.getloc())} \) evaluates to \( \text{true} \), in which case the function first increments \( \text{fmt} \) until \( \text{fmt} == \text{fmtend} \mid \text{isspace(*fmt, f.getloc())} \) evaluates to \( \text{true} \), then advances \( \text{s} \) until \( \text{s} == \text{end} \mid \text{isspace(*s, f.getloc())} \) is \( \text{true} \), and finally resumes looping.

6. The next character read from \( \text{s} \) matches the element pointed to by \( \text{fmt} \) in a case-insensitive comparison, in which case the function evaluates \( \text{++fmt, ++s} \) and continues looping. Otherwise, the function evaluates \( \text{err} = \text{ios\_base::failbit} \).

[Note 1: The function uses the \( \text{ctype<charT>} \) facet installed in \( \text{f}'s \) locale to determine valid whitespace characters. It is unspecified by what means the function performs case-insensitive comparison or whether multi-character sequences are considered while doing so. — end note]

Returns: \( \text{s} \).

### 28.4.6.2.3 Virtual functions

#### dateorder do\_date\_order() const;
Returns: An enumeration value indicating the preferred order of components for those date formats that are composed of day, month, and year.\(^{277}\) Returns \( \text{no\_order} \) if the date format specified by ‘\x’ contains other variable components (e.g., Julian day, week number, week day).

#### iter\_type do\_get\_time(iter\_type s, iter\_type end, ios\_base& str, ios\_base::iostate& err, tm* t) const;
Effects: Reads characters starting at \( s \) until it has extracted those \( \text{struct tm} \) members, and remaining format characters, used by \( \text{time\_put<>::put} \) to produce the format specified by ‘\%H:\%M:\%S’, or until it encounters an error or end of sequence.
Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid time.

#### iter\_type do\_get\_date(iter\_type s, iter\_type end, ios\_base& str, ios\_base::iostate& err, tm* t) const;
Effects: Reads characters starting at \( s \) until it has extracted those \( \text{struct tm} \) members and remaining format characters used by \( \text{time\_put<>::put} \) to produce one of the following formats, or until it encounters an error. The format depends on the value returned by \( \text{date\_order()} \) as shown in Table 113.
An implementation may also accept additional implementation-defined formats.
Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid date.

\(^{277}\) This function is intended as a convenience only, for common formats, and can return \( \text{no\_order} \) in valid locales.
Table 113: do_get_date effects [tab:locale.time.get.dogetdate]

<table>
<thead>
<tr>
<th>date_order()</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_order</td>
<td>&quot;%m%d%y&quot;</td>
</tr>
<tr>
<td>dmy</td>
<td>&quot;%d%m%y&quot;</td>
</tr>
<tr>
<td>mdy</td>
<td>&quot;%m%d%y&quot;</td>
</tr>
<tr>
<td>ymd</td>
<td>&quot;%y%m%d&quot;</td>
</tr>
<tr>
<td>ydm</td>
<td>&quot;%y%d%m&quot;</td>
</tr>
</tbody>
</table>

iter_type do_get_weekday(iter_type s, iter_type end, ios_base& str, 
                         ios_base::iostate& err, tm* t) const;
iter_type do_get_monthname(iter_type s, iter_type end, ios_base& str, 
                           ios_base::iostate& err, tm* t) const;

Effects: Reads characters starting at s until it has extracted the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full name, it continues reading until it matches the full name or fails. It sets the appropriate struct tm member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid name.

iter_type do_get_year(iter_type s, iter_type end, ios_base& str, 
                      ios_base::iostate& err, tm* t) const;

Effects: Reads characters starting at s until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the t->tm_year member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

iter_type do_get(iter_type s, iter_type end, ios_base& f, 
                 ios_base::iostate& err, tm* t, char format, char modifier) const;

Preconditions: t points to an object.

Effects: The function starts by evaluating err = ios_base::goodbit. It then reads characters starting at s until it encounters an error, or until it has extracted and assigned those struct tm members, and any remaining format characters, corresponding to a conversion directive appropriate for the ISO/IEC 9945 function strptime, formed by concatenating '%', the modifier character, when non-NUL, and the format character. When the concatenation fails to yield a complete valid directive the function leaves the object pointed to by t unchanged and evaluates err |= ios_base::failbit. When s == end evaluates to true after reading a character the function evaluates err |= ios_base::eofbit. For complex conversion directives such as %c, %x, or %X, or directives that involve the optional modifiers E or O, when the function is unable to unambiguously determine some or all struct tm members from the input sequence [s, end], it evaluates err |= ios_base::eofbit. In such cases the values of those struct tm members are unspecified and may be outside their valid range.

Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid input sequence for the given format and modifier.

Remarks: It is unspecified whether multiple calls to do_get() with the address of the same struct tm object will update the current contents of the object or simply overwrite its members. Portable programs should zero out the object before invoking the function.

28.4.6.3 Class template time_get_byname [locale.time.getbyname]

namespace std {
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class time_get_byname : public time_get<charT, InputIterator> {
        public:
            using dateorder = time_base::dateorder;
            using iter_type = InputIterator;

§ 28.4.6.3 1365
explicit time_getbyname(const char*, size_t refs = 0);
explicit time_getbyname(const string&, size_t refs = 0);

protected:
- time_getbyname();
};

28.4.6.4 Class template time_put

namespace std {
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
  class time_put : public locale::facet {
  public:
    using char_type = charT;
    using iter_type = OutputIterator;

    explicit time_put(size_t refs = 0);

    // the following is implemented in terms of other member functions.
    iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
                  const charT* pattern, const charT* pat_end) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  const tm* tmb, char format, char modifier = 0) const;

  protected:
    - time_put();
    virtual iter_type do_put(iter_type s, ios_base&, char_type, const tm* t,
                             char format, char modifier) const;
  };
}

28.4.6.4.1 Members

iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              char format, char modifier = 0) const;

Effects: The first form steps through the sequence from pattern to pat_end, identifying characters that are part of a format sequence. Each character that is not part of a format sequence is written to s immediately, and each format sequence, as it is identified, results in a call to do_put; thus, format elements and other characters are interleaved in the output in the order in which they appear in the pattern. Format sequences are identified by converting each character c to a char value as if by ct.narrow(c, 0), where ct is a reference to ctype<charT> obtained from str.getloc(). The first character of each sequence is equal to '%', followed by an optional modifier character mod and a format specifier character spec as defined for the function strftime. If no modifier character is present, mod is zero. For each valid format sequence identified, calls do_put(s, str, fill, t, spec, mod).

The second form calls do_put(s, str, fill, t, format, modifier).

[Note 1: The fill argument can be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — end note]

Returns: An iterator pointing immediately after the last character produced.

28.4.6.4.2 Virtual functions

iter_type do_put(iter_type s, ios_base&, char_type fill, const tm* t,
                 char format, char modifier) const;

Effects: Formats the contents of the parameter t into characters placed on the output sequence s. Formatting is controlled by the parameters format and modifier, interpreted identically as the format

278) Although the C programming language defines no modifiers, most vendors do.
specifiers in the string argument to the standard library function \texttt{strftime()}, except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.

\textit{[Note 1: Interpretation of the \texttt{modifier} argument is implementation-defined. — end note]}

2 \textbf{Returns:} An iterator pointing immediately after the last character produced.

\textit{[Note 2: The \texttt{fill} argument can be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — end note]}

3 \textbf{Recommended practice:} Interpretation of the \texttt{modifier} should follow POSIX conventions. Implementations should refer to other standards such as POSIX for a specification of the character sequences produced for those specifiers described as depending on the C locale.

28.4.6.5 Class template \texttt{time\_put\_byname}

namespace std {
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
  class time_put_byname : public time_put<charT, OutputIterator> {
    public:
      using char_type = charT;
      using iter_type = OutputIterator;
      explicit time_put_byname(const char*, size_t refs = 0);
      explicit time_put_byname(const string&, size_t refs = 0);
    protected:
      ~time_put_byname();
  };
}

28.4.7 The monetary category

28.4.7.1 General

1 These templates handle monetary formats. A template parameter indicates whether local or international monetary formats are to be used.

2 All specifications of member functions for \texttt{money\_put} and \texttt{money\_get} in the subclauses of 28.4.7 only apply to the specializations required in Tables 102 and 103 (28.3.1.2.1). Their members use their \texttt{ios\_base\&}, \texttt{ios\_base::iostate\&}, and \texttt{fill} arguments as described in 28.4, and the \texttt{moneypunct<>} and \texttt{ctype<>} facets, to determine formatting details.

28.4.7.2 Class template \texttt{money\_get}

namespace std {
  template<class charT, class InputIterator = istreambuf_iterator<charT>>
  class money_get : public locale::facet {
    public:
      using char_type = charT;
      using iter_type = InputIterator;
      using string_type = basic_string<charT>;
      explicit money_get(size_t refs = 0);
      iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, long double& units) const;
      iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, string_type& digits) const;
    static locale::id id;
    protected:
      ~money_get();
      virtual iter_type do_get(iter_type, iter_type, bool, ios_base&, ios_base::iostate& err, long double& units) const;
  };
}
virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,_ios_base::iostate& err, string_type& digits) const;
};

28.4.7.2.1 Members

iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, long double& quant) const;
iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, string_type& quant) const;

Returns: do_get(s, end, intl, f, err, quant).

28.4.7.2.2 Virtual functions

iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& str, ios_base::iostate& err, long double& units) const;
iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& str, ios_base::iostate& err, string_type& digits) const;

Effects: Reads characters from s to parse and construct a monetary value according to the format specified by a moneypunct<charT, Intl> facet reference mp and the character mapping specified by a ctype<charT> facet reference ct obtained from the locale returned by str.getloc(), and str.flags(). If a valid sequence is recognized, does not change err; otherwise, sets err to (err|str.failbit), or (err|str.failbit|str.eofbit) if no more characters are available, and does not change units or digits. Uses the pattern returned by mp.neg_format() to parse all values. The result is returned as an integral value stored in units or as a sequence of digits possibly preceded by a minus sign (as produced by ct.widen(c) where c is '-' or in the range from '0' through '9' (inclusive)) stored in digits.

[Example 1: The sequence $1,056.23 in a common United States locale would yield, for units, 105623, or, for digits, "105623". — end example]

If mp.grouping() indicates that no thousands separators are permitted, any such characters are not read, and parsing is terminated at the point where they first appear. Otherwise, thousands separators are optional; if present, they are checked for correct placement only after all format components have been read.

Where money_base::space or money_base::none appears as the last element in the format pattern, no whitespace is consumed. Otherwise, where money_base::space appears in any of the initial elements of the format pattern, at least one whitespace character is required. Where money_base::none appears in any of the initial elements of the format pattern, white space is allowed but not required. If (str.flags() & str.showbase) is false, the currency symbol is optional and is consumed only if other characters are needed to complete the format; otherwise, the currency symbol is required.

If the first character (if any) in the string pos returned by mp.positive_sign() or the string neg returned by mp.negative_sign() is recognized in the position indicated by sign in the format pattern, it is consumed and any remaining characters in the string are required after all the other format components.

[Example 2: If showbase is off, then for a neg value of "()" and a currency symbol of "L", in "(100 L)" the "L" is consumed; but if neg is "-", the "L" in "-100 L" is not consumed. — end example]

If pos or neg is empty, the sign component is optional, and if no sign is detected, the result is given the sign that corresponds to the source of the empty string. Otherwise, the character in the indicated position must match the first character of pos or neg, and the result is given the corresponding sign. If the first character of pos is equal to the first character of neg, or if both strings are empty, the result is given a positive sign.

Digits in the numeric monetary component are extracted and placed in digits, or into a character buffer buf1 for conversion to produce a value for units, in the order in which they appear, preceded by a minus sign if and only if the result is negative. The value units is produced as if by:

```cpp
for (int i = 0; i < n; ++i)
    buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms];
```

279) The semantics here are different from ct.narrow.
where *n* is the number of characters placed in *buf1*, *buf2* is a character buffer, and the values *src* and *atoms* are defined as if by

```
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
ct.widen(src, src + sizeof(src) - 1, atoms);
```

*Returns:* An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

### 28.4.7.3 Class template money_put

```
namespace std {
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class money_put : public locale::facet {
        public:
            using char_type = charT;
            using iter_type = OutputIterator;
            using string_type = basic_string<charT>;

            explicit money_put(size_t refs = 0);

            iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, long double units) const;
            iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, const string_type& digits) const;

            static locale::id id;

        protected:
            ~money_put();
            virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill, long double units) const;
            virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill, const string_type& digits) const;
    };
}
```

#### 28.4.7.3.1 Members

```
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, long double quant) const;
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, const string_type& quant) const;
```

*Returns:* `do_put(s, intl, f, loc, quant)`.

#### 28.4.7.3.2 Virtual functions

```
iter_type do_put(iter_type s, bool intl, ios_base& str, char_type fill, long double units) const;
iter_type do_put(iter_type s, bool intl, ios_base& str, char_type fill, const string_type& digits) const;
```

*Effects:* Writes characters to *s* according to the format specified by a `moneypunct<charT, Intl>` facet reference `mp` and the character mapping specified by a `ctype<charT>` facet reference `ct` obtained from the locale returned by `str.getloc()`, and `str.flags()`. The argument `units` is transformed into a sequence of wide characters as if by

```
ct.widen(buf1, buf1 + sprintf(buf1, "%Lf", units), buf2)
```

for character buffers *buf1* and *buf2*. If the first character in *digits* or *buf2* is equal to `ct.widen('-')`, then the pattern used for formatting is the result of `mp.neg_format()`; otherwise the pattern is the result of `mp.pos_format()`. Digit characters are written, interspersed with any thousands separators and decimal point specified by the format, in the order they appear (after the optional leading minus sign) in *digits* or *buf2*. In *digits*, only the optional leading minus sign and the immediately subsequent digit
characters (as classified according to ct) are used; any trailing characters (including digits appearing after a non-digit character) are ignored. Calls str.width(0).

Returns: An iterator pointing immediately after the last character produced.

Remarks: The currency symbol is generated if and only if (str.flags() & str.showbase) is nonzero. If the number of characters generated for the specified format is less than the value returned by str.width() on entry to the function, then copies of fill are inserted as necessary to pad to the specified width. For the value af equal to (str.flags() & str.adjustfield), if (af == str.internal) is true, the fill characters are placed where none or space appears in the formatting pattern; otherwise if (af == str.left) is true, they are placed after the other characters; otherwise, they are placed before the other characters.

[Note 1: It is possible, with some combinations of format patterns and flag values, to produce output that cannot be parsed using num_get<>::get. —end note]

28.4.7.4 Class template moneypunct

28.4.7.4.1 General

namespace std {

    class money_base {
    public:
        enum part { none, space, symbol, sign, value };
        struct pattern { char field[4]; };
    }

    template<class charT, bool International = false>
    class moneypunct : public locale::facet, public money_base {
    public:
        using char_type = charT;
        using string_type = basic_string<charT>;

        explicit moneypunct(size_t refs = 0);

        charT decimal_point() const;
        charT thousands_sep() const;
        string grouping() const;
        string_type curr_symbol() const;
        string_type positive_sign() const;
        string_type negative_sign() const;
        int frac_digits() const;
        pattern pos_format() const;
        pattern neg_format() const;

        static locale::id id;
        static const bool intl = International;

    protected:
        ~moneypunct();
        virtual charT do_decimal_point() const;
        virtual charT do_thousands_sep() const;
        virtual string do_grouping() const;
        virtual string_type do_curr_symbol() const;
        virtual string_type do_positive_sign() const;
        virtual string_type do_negative_sign() const;
        virtual int do_frac_digits() const;
        virtual pattern do_pos_format() const;
        virtual pattern do_neg_format() const;
    };
}

1 The moneypunct<> facet defines monetary formatting parameters used by money_get<> and money_put<>. A monetary format is a sequence of four components, specified by a pattern value p, such that the part value static_cast<part>(p.field[i]) determines the i\textsuperscript{th} component of the format\textsuperscript{280} In the field member of

\textsuperscript{280} An array of char, rather than an array of part, is specified for pattern::field purely for efficiency.
a pattern object, each value symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last.

Where none or space appears, whitespace is permitted in the format, except where none appears at the end, in which case no whitespace is permitted. The value space indicates that at least one space is required at that position. Where symbol appears, the sequence of characters returned by curr_symbol() is permitted, and can be required. Where sign appears, the first (if any) of the sequence of characters returned by positive_sign() or negative_sign() (respectively as the monetary value is non-negative or negative) is required. Any remaining characters of the sign sequence are required after all other format components. Where value appears, the absolute numeric monetary value is required.

The format of the numeric monetary value is a decimal number:

\[ \text{value: units fractional_opt decimal-point digits} \]
\[ \text{fractional: decimal-point digits_opt} \]

if frac_digits() returns a positive value, or

\[ \text{value: units} \]

otherwise. The symbol decimal-point indicates the character returned by decimal_point(). The other symbols are defined as follows:

\[ \text{units: digits} \]
\[ \text{digits: adigit digits} \]
\[ \text{thousands-sep units} \]
\[ \text{adigit digits_opt} \]

In the syntax specification, the symbol adigit is any of the values ct.widen(c) for c in the range '0' through '9' (inclusive) and ct is a reference of type const ctype<charT>& obtained as described in the definitions of money_get<> and money_put<>. The symbol thousands-sep is the character returned by thousands_sep(). The space character used is the value ct.widen(' '). White space characters are those characters c for which ci.is(space, c) returns true. The number of digits required after the decimal point (if any) is exactly the value returned by frac_digits().

The placement of thousands-separator characters (if any) is determined by the value returned by grouping(), defined identically as the member numpunct<>::do_grouping().

28.4.7.4.2 Members [locale.moneypunct.members]

charT decimal_point() const;
charT thousands_sep() const;
string grouping() const;
string_type curr_symbol() const;
string_type positive_sign() const;
string_type negative_sign() const;
int frac_digits() const;
pattern pos_format() const;
pattern neg_format() const;

1 Each of these functions \( F \) returns the result of calling the corresponding virtual member function do_\( F \).  

28.4.7.4.3 Virtual functions [locale.moneypunct.virtuals]

charT do_decimal_point() const;

1 Returns: The radix separator to use in case do_frac_digits() is greater than zero.\(^{281}\)

charT do_thousands_sep() const;

2 Returns: The digit group separator to use in case do_grouping() specifies a digit grouping pattern.\(^{282}\)

\(^{281}\) In common U.S. locales this is \('.\').

\(^{282}\) In common U.S. locales this is \',''.

§ 28.4.7.4.3 1371
string do_grouping() const;

Returns: A pattern defined identically as, but not necessarily equal to, the result of numpunct<charT>::
do_grouping().

string_type do_curr_symbol() const;

Returns: A string to use as the currency identifier symbol.

[Note 1: For specializations where the second template parameter is true, this is typically four characters long:
a three-letter code as specified by ISO 4217 followed by a space. — end note]

string_type do_positive_sign() const;
string_type do_negative_sign() const;

Returns: do_positive_sign() returns the string to use to indicate a positive monetary value; do_negative_sign() returns the string to use to indicate a negative value.

int do_fractions() const;

Returns: The number of digits after the decimal radix separator, if any.

pattern do_pos_format() const;
pattern do_neg_format() const;

Returns: The specializations required in Table 103 (28.3.1.2.1), namely

— moneypunct<char>,
— moneypunct<wchar_t>,
— moneypunct<char, true>, and
— moneypunct<wchar_t, true>,

return an object of type pattern initialized to { symbol, sign, none, value }.

28.4.7.5 Class template moneypunct_byname [locale.moneypunct.byname]

namespace std {
    template<class charT, bool Intl = false>
    class moneypunct_byname : public moneypunct<charT, Intl> {
        public:
            using pattern = money_base::pattern;
            using string_type = basic_string<charT>;

            explicit moneypunct_byname(const char*, size_t refs = 0);
            explicit moneypunct_byname(const string&, size_t refs = 0);

        protected:
            ~moneypunct_byname();
        }
    }

28.4.8 The message retrieval category [category.messages]

28.4.8.1 General [category.messages.general]

Class messages<charT> implements retrieval of strings from message catalogs.

28.4.8.2 Class template messages [locale.messages]

28.4.8.2.1 General [locale.messages.general]

namespace std {
    class messages_base {
        public:
            using catalog = unspecified signed integer type;
    }

283) To specify grouping by 3s, the value is "\003" not "3".
284) This is usually the empty string.
285) In common U.S. locales, this is 2.
286) Note that the international symbol returned by do_curr_symbol() usually contains a space, itself; for example, "USD ".

§ 28.4.8.2.1
template<class charT>
    class messages : public locale::facet, public messages_base {
        public:
            using char_type = charT;
            using string_type = basic_string<charT>;
            explicit messages(size_t refs = 0);
            catalog open(const string& fn, const locale&) const;
            string_type get(catalog c, int set, int msgid,
                const string_type& dfault) const;
            void close(catalog c) const;
            static locale::id id;
        protected:
            ~messages();
            virtual catalog do_open(const string&, const locale&) const;
            virtual string_type do_get(catalog, int set, int msgid,
                const string_type& dfault) const;
            virtual void do_close(catalog) const;
    };

    Values of type messages_base::catalog usable as arguments to members get and close can be obtained only by calling member open.

28.4.8.2.2 Members

    catalog open(const string& name, const locale& loc) const;
        Returns: do_open(name, loc).

    string_type get(catalog cat, int set, int msgid, const string_type& dfault) const;
        Returns: do_get(cat, set, msgid, dfault).

    void close(catalog cat) const;
        Effects: Calls do_close(cat).

28.4.8.2.3 Virtual functions

    catalog do_open(const string& name, const locale& loc) const;
        Returns: A value that may be passed to get() to retrieve a message from the message catalog identified by the string name according to an implementation-defined mapping. The result can be used until it is passed to close().
        Returns a value less than 0 if no such catalog can be opened.
        Remarks: The locale argument loc is used for character set code conversion when retrieving messages, if needed.

    string_type do_get(catalog cat, int set, int msgid, const string_type& dfault) const;
        Preconditions: cat is a catalog obtained from open() and not yet closed.
        Returns: A message identified by arguments set, msgid, and dfault, according to an implementation-defined mapping. If no such message can be found, returns dfault.

    void do_close(catalog cat) const;
        Preconditions: cat is a catalog obtained from open() and not yet closed.
        Effects: Releases unspecified resources associated with cat.
        Remarks: The limit on such resources, if any, is implementation-defined.
28.4.8.3 Class template messages_byname

namespace std {
    template<class charT>
    class messages_byname : public messages<charT> {
        public:
            using catalog = messages_base::catalog;
            using string_type = basic_string<charT>;

            explicit messages_byname(const char*, size_t refs = 0);
            explicit messages_byname(const string&, size_t refs = 0);

        protected:
            ~messages_byname();
    };
}

28.5 C library locales

28.5.1 Header <clocale> synopsis

namespace std {
    struct lconv;

    char* setlocale(int category, const char* locale);
    lconv* localeconv();
}

#define NULL see 17.2.3
#define LC_ALL see below
#define LC_COLLATE see below
#define LC_CTYPE see below
#define LC_MONETARY see below
#define LC_NUMERIC see below
#define LC_TIME see below

1 The contents and meaning of the header <clocale> are the same as the C standard library header <locale.h>.

28.5.2 Data races

1 Calls to the function setlocale may introduce a data race (16.4.6.10) with other calls to setlocale or with calls to the functions listed in Table 114.

See also: ISO C 7.11

Table 114: Potential setlocale data races

<table>
<thead>
<tr>
<th>fprintf</th>
<th>isprint</th>
<th>iswdigit</th>
<th>localeconv</th>
<th>tolower</th>
</tr>
</thead>
<tbody>
<tr>
<td>fscanf</td>
<td>ispunct</td>
<td>iswgraph</td>
<td>mblen</td>
<td>toupper</td>
</tr>
<tr>
<td>isalnum</td>
<td>isspace</td>
<td>iswlower</td>
<td>mbstowcs</td>
<td>tolower</td>
</tr>
<tr>
<td>isalpha</td>
<td>isupper</td>
<td>iswprint</td>
<td>mbtowc</td>
<td>towupper</td>
</tr>
<tr>
<td>isblank</td>
<td>iswalpha</td>
<td>iswspace</td>
<td>strcoll</td>
<td>wcstod</td>
</tr>
<tr>
<td>iscntrl</td>
<td>iswctrl</td>
<td>iswxdigit</td>
<td>strndir</td>
<td>wcsxfrm</td>
</tr>
<tr>
<td>isdigit</td>
<td>iswblank</td>
<td>iswupper</td>
<td>strerror</td>
<td>wcstombs</td>
</tr>
<tr>
<td>isgraph</td>
<td>iswctrl</td>
<td>isxdigit</td>
<td>strtod</td>
<td>wcscoll</td>
</tr>
<tr>
<td>islower</td>
<td>iswctype</td>
<td>isxdigit</td>
<td>strxfrm</td>
<td>wcstomb</td>
</tr>
</tbody>
</table>
29 Input/output library [input.output]

29.1 General [input.output.general]

1 This Clause describes components that C++ programs may use to perform input/output operations.

2 The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 115.

Table 115: Input/output library summary [tab:iostreams.summary]

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.2 Requirements</td>
<td>&lt;iosfwd&gt;</td>
</tr>
<tr>
<td>29.3 Forward declarations</td>
<td>&lt;iostream&gt;</td>
</tr>
<tr>
<td>29.4 Standard istream objects</td>
<td>&lt;ios&gt;</td>
</tr>
<tr>
<td>29.5 Iostreams base classes</td>
<td>&lt;streambuf&gt;</td>
</tr>
<tr>
<td>29.6 Stream buffers</td>
<td>&lt;iostream&gt;, &lt;ostream&gt;</td>
</tr>
<tr>
<td>29.7 Formatting and manipulators</td>
<td>&lt;iostream&gt;, &lt;ostream&gt;</td>
</tr>
<tr>
<td>29.8 String streams</td>
<td>&lt;iostream&gt;</td>
</tr>
<tr>
<td>29.9 File streams</td>
<td>&lt;fstream&gt;</td>
</tr>
<tr>
<td>29.10 Synchronized output streams</td>
<td>&lt;fstream&gt;</td>
</tr>
<tr>
<td>29.11 File systems</td>
<td>&lt;filesystem&gt;</td>
</tr>
<tr>
<td>29.12 C library files</td>
<td>&lt;cstdio&gt;, &lt;cinttypes&gt;</td>
</tr>
</tbody>
</table>

3 [Note 1: Figure 7 illustrates relationships among various types described in this Clause. A line from A to B indicates that A is an alias (e.g., a typedef) for B or that A is defined in terms of B.]

Figure 7: Stream position, offset, and size types [fig:iostreams.streampos]

— end note]

29.2 Iostreams requirements [iostreams.requirements]

29.2.1 Imbue limitations [iostreams.requirements.imbue]

1 No function described in Clause 29 except for ios_base::imbue and basic_filebuf::pubimbue causes any instance of basic_ios::imbue or basic_streambuf::imbue to be called. If any user function called from a function declared in Clause 29 or as an overriding virtual function of any class declared in Clause 29 calls imbue, the behavior is undefined.
29.2.2 Positioning type limitations

The classes of Clause 29 with template arguments charT and traits behave as described if traits::pos_type and traits::off_type are streampos and streamoff respectively. Except as noted explicitly below, their behavior when traits::pos_type and traits::off_type are other types is implementation-defined.

In the classes of Clause 29, a template parameter with name charT represents a member of the set of types containing char, wchar_t, and any other implementation-defined character types that meet the requirements for a character on which any of the iostream components can be instantiated.

29.2.3 Thread safety

Concurrent access to a stream object (29.8, 29.9), stream buffer object (29.6), or C Library stream (29.12) by multiple threads may result in a data race (6.9.2) unless otherwise specified (29.4).

[Note 1: Data races result in undefined behavior (6.9.2). — end note]

If one thread makes a library call a that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call b such that this does not result in a data race, then a’s write synchronizes with b’s read.

29.3 Forward declarations

29.3.1 Header <iosfwd> synopsis

namespace std {
    template<class charT> struct char_traits;
    template<> struct char_traits<char>;
    template<> struct char_traits<char8_t>;
    template<> struct char_traits<char16_t>;
    template<> struct char_traits<char32_t>;
    template<> struct char_traits<wchar_t>;
    template<class T> class allocator;

    template<class charT, class traits = char_traits<charT>>
    class basic_ion;
    template<class charT, class traits = char_traits<charT>>
    class basic_streambuf;
    template<class charT, class traits = char_traits<charT>>
    class basic_istream;
    template<class charT, class traits = char_traits<charT>>
    class basic_ostream;
    template<class charT, class traits = char_traits<charT>>
    class basic_iostream;
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_stringbuf;
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_stringstream;
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_filebuf;
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_ifstream;
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_ofstream;
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_fstream;

§ 29.3.1
template<class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT>>
class basic_syncbuf;

template<class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT>>
class basic_osyncstream;

template<class charT, class traits = char_traits<charT>>
class istreambuf_iterator;

template<class charT, class traits = char_traits<charT>>
class ostreambuf_iterator;

using ios = basic_ios<char>;
using wios = basic_ios<wchar_t>;

using streambuf = basic_streambuf<char>;
using istream = basic_istream<char>;
using ostream = basic_ostream<char>;
using iostream = basic_iostream<char>;

using stringbuf = basic_stringbuf<char>;
using istrstream = basic_istreamstream<char>;
using ostrstream = basic_ostringstream<char>;
using stringstream = basic_stringstream<char>;

using filebuf = basic_filebuf<char>;
using ifstream = basic_ifstream<char>;
using ofstream = basic_ofstream<char>;
using fstream = basic_fstream<char>;

using syncbuf = basic_syncbuf<char>;
using osyncstream = basic_osyncstream<char>;

using u8streambuf = basic_streambuf<char8_t>;
using u16streambuf = basic_streambuf<char16_t>;
using u32streambuf = basic_streambuf<char32_t>;

using streampos = fpos<char_traits<char>::state_type>;
using wstreampos = fpos<char_traits<wchar_t>::state_type>;
using u8streampos = fpos<char_traits<char8_t>::state_type>;
using u16streampos = fpos<char_traits<char16_t>::state_type>;
using u32streampos = fpos<char_traits<char32_t>::state_type>;

1 Default template arguments are described as appearing both in <iosfwd> and in the synopsis of other headers but it is well-formed to include both <iosfwd> and one or more of the other headers.

287 It is the implementation’s responsibility to implement headers so that including <iosfwd> and other headers does not violate the rules about multiple occurrences of default arguments.
29.3.2 Overview

The class template specialization `basic_ios<charT, traits>` serves as a virtual base class for the class templates `basic_istream`, `basic_ostream`, and class templates derived from them. `basic_iosstream` is a class template derived from both `basic_istream<charT, traits>` and `basic_ostream<charT, traits>`.

The class template specialization `basic_streambuf<charT, traits>` serves as a base class for class templates `basic_stringbuf`, `basic_filebuf`, and `basic_syncbuf`.

The class template specialization `basic_istream<charT, traits>` serves as a base class for class templates `basic_istringstream` and `basic_ifstream`.

The class template specialization `basic_ostream<charT, traits>` serves as a base class for class templates `basic_ostringstream`, `basic_ofstream`, and `basic_osyncstream`.

The class template specialization `basic_iostream<charT, traits>` serves as a base class for class templates `basic_stringstream` and `basic_fstream`.

[Note 1: For each of the class templates above, the program is ill-formed if `traits::char_type` is not the same type as `charT` (21.2). —end note]

Other typedef-names define instances of class templates specialized for `char` or `wchar_t` types.

Specializations of the class template `fpos` are used for specifying file position information.

[Example 1: The types `streampos` and `wstreampos` are used for positioning streams specialized on `char` and `wchar_t` respectively. —end example]

[Note 2: This synopsis suggests a circularity between `streampos` and `char_traits<char>`. An implementation can avoid this circularity by substituting equivalent types. —end note]

29.4 Standard iostream objects

29.4.1 Header `<iostream>` synopsis

```cpp
#include <ios> // see 29.5.1
#include <streambuf> // see 29.6.1
#include <iostream> // see 29.7.1
#include <ostream> // see 29.7.2

namespace std {
  extern istream cin;
  extern ostream cout;
  extern ostream cerr;
  extern ostream clog;
  extern wistream wcin;
  extern wostream wcout;
  extern wostream wcerr;
  extern wostream wclog;
}
```

29.4.2 Overview

In this Clause, the type name `FILE` refers to the type `FILE` declared in `<stdio>` (29.12.1).

The header `<iostream>` declares objects that associate objects with the standard C streams provided for by the functions declared in `<stdio>`, and includes all the headers necessary to use these objects.

The objects are constructed and the associations are established at some time prior to or during the first time an object of class `ios_base::Init` is constructed, and in any case before the body of `main` (6.9.3.1) begins execution. The objects are not destroyed during program execution.288

Recommended practice: If it is possible for them to do so, implementations should initialize the objects earlier than required.

The results of including `<iostream>` in a translation unit shall be as if `<iostream>` defined an instance of `ios_base::Init` with static storage duration.

---

288) Constructors and destructors for objects with static storage duration can access these objects to read input from `stdin` or write output to `stdout` or `stderr`. 

§ 29.4.2 1378
Mixing operations on corresponding wide- and narrow-character streams follows the same semantics as mixing such operations on FILEs, as specified in the C standard library.

Concurrent access to a synchronized (29.5.3.5) standard iostream object’s formatted and unformatted input (29.7.4.2) and output (29.7.5.2) functions or a standard C stream by multiple threads does not result in a data race (6.9.2).

[Note 1: Unsynchronized concurrent use of these objects and streams by multiple threads can result in interleaved characters. — end note]

See also: ISO C 7.21.2

29.4.3 Narrow stream objects

```cpp
istream cin;
```

The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>` (29.12.1).

After the object `cin` is initialized, `cin.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (29.5.5.2).

```cpp
ostream cout;
```

The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (29.12.1).

```cpp
ostream cerr;
```

The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

After the object `cerr` is initialized, `cerr.flags() & unitbuf` is nonzero and `cerr.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (29.5.5.2).

```cpp
ostream clog;
```

The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

29.4.4 Wide stream objects

```cpp
wistream wcin;
```

The object `wcin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>` (29.12.1).

After the object `wcin` is initialized, `wcin.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (29.5.5.2).

```cpp
wostream wcout;
```

The object `wcout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (29.12.1).

```cpp
wostream wcerr;
```

The object `wcerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

After the object `wcerr` is initialized, `wcerr.flags() & unitbuf` is nonzero and `wcerr.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (29.5.5.2).

```cpp
wostream wclog;
```

The object `wclog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

29.5 Iostreams base classes

29.5.1 Header `<ios>` synopsis

```cpp
#include <iosfwd>
```

// see 29.3.1
namespace std {
  using streamoff = implementation-defined;
  using streamsize = implementation-defined;
  template<class stateT> class fpos;

  class ios_base;
  template<class charT, class traits = char_traits<charT>>
  class basic_ios;

  // 29.5.6, manipulators
  ios_base& boolalpha (ios_base& str);
  ios_base& noboolalpha(ios_base& str);

  ios_base& showbase (ios_base& str);
  ios_base& noshowbase (ios_base& str);

  ios_base& showpoint (ios_base& str);
  ios_base& noshowpoint(ios_base& str);

  ios_base& showpos (ios_base& str);
  ios_base& noshowpos (ios_base& str);

  ios_base& skipws (ios_base& str);
  ios_base& noskipws (ios_base& str);

  ios_base& uppercase (ios_base& str);
  ios_base& nouppercase(ios_base& str);

  ios_base& unitbuf (ios_base& str);
  ios_base& nounitbuf (ios_base& str);

  // 29.5.6.2, adjustfield
  ios_base& internal (ios_base& str);
  ios_base& left (ios_base& str);
  ios_base& right (ios_base& str);

  // 29.5.6.3, basefield
  ios_base& dec (ios_base& str);
  ios_base& hex (ios_base& str);
  ios_base& oct (ios_base& str);

  // 29.5.6.4, floatfield
  ios_base& fixed (ios_base& str);
  ios_base& scientific (ios_base& str);
  ios_base& hexfloat (ios_base& str);
  ios_base& defaultfloat(ios_base& str);

  // 29.5.7, error reporting
  enum class io_errc {
    stream = 1
  };

  template<> struct is_error_code_enum<io_errc> : public true_type { }; error_code make_error_code(io_errc e) noexcept;
  error_condition make_error_condition(io_errc e) noexcept;
  const error_category& iostream_category() noexcept;
}

29.5.2 Types                                  [stream.types]
using streamoff = implementation-defined;

1 The type streamoff is a synonym for one of the signed basic integral types of sufficient size to represent
the maximum possible file size for the operating system.289

using streamsize = implementation-defined;

2 The type streamsize is a synonym for one of the signed basic integral types. It is used to represent the number of characters transferred in an I/O operation, or the size of I/O buffers.290

29.5.3 Class ios_base

namespace std {

class ios_base { public:

    class failure; // see below

    // 29.5.3.2.2, fmtflags
    using fmtflags = T1;
    static constexpr fmtflags boolalpha = unspecified;
    static constexpr fmtflags dec = unspecified;
    static constexpr fmtflags fixed = unspecified;
    static constexpr fmtflags hex = unspecified;
    static constexpr fmtflags internal = unspecified;
    static constexpr fmtflags left = unspecified;
    static constexpr fmtflags oct = unspecified;
    static constexpr fmtflags right = unspecified;
    static constexpr fmtflags scientific = unspecified;
    static constexpr fmtflags showbase = unspecified;
    static constexpr fmtflags showpoint = unspecified;
    static constexpr fmtflags showpos = unspecified;
    static constexpr fmtflags skipws = unspecified;
    static constexpr fmtflags unitbuf = unspecified;
    static constexpr fmtflags uppercase = unspecified;
    static constexpr fmtflags adjustfield = see below;
    static constexpr fmtflags basefield = see below;
    static constexpr fmtflags floatfield = see below;

    // 29.5.3.2.3, iostate
    using iostate = T2;
    static constexpr iostate badbit = unspecified;
    static constexpr iostate eofbit = unspecified;
    static constexpr iostate failbit = unspecified;
    static constexpr iostate goodbit = see below;

    // 29.5.3.2.4, openmode
    using openmode = T3;
    static constexpr openmode app = unspecified;
    static constexpr openmode ate = unspecified;
    static constexpr openmode binary = unspecified;
    static constexpr openmode in = unspecified;
    static constexpr openmode out = unspecified;
    static constexpr openmode trunc = unspecified;

    // 29.5.3.2.5, seekdir
    using seekdir = T4;
    static constexpr seekdir beg = unspecified;
    static constexpr seekdir cur = unspecified;
    static constexpr seekdir end = unspecified;

    class Init;

289) Typically long long.
290) streamsize is used in most places where ISO C would use size_t.
ios_base defines several member types:

1. a type failure, defined as either a class derived from system_error or a synonym for a class derived from system_error;
2. a class Init;
3. three bitmask types, fmtflags, iostate, and openmode;
4. an enumerated type, seekdir.

It maintains several kinds of data:

1. state information that reflects the integrity of the stream buffer;
2. control information that influences how to interpret (format) input sequences and how to generate (format) output sequences;
3. additional information that is stored by the program for its private use.

Note 1: For the sake of exposition, the maintained data is presented here as:

1. static int index, specifies the next available unique index for the integer or pointer arrays maintained for the private use of the program, initialized to an unspecified value;
— long* iarray, points to the first element of an arbitrary-length long array maintained for the private use of the program;

— void** parray, points to the first element of an arbitrary-length pointer array maintained for the private use of the program.

—end note]

29.5.3.2 Types

29.5.3.2.1 Class ios_base::failure

namespace std {

  class ios_base::failure : public system_error {
  public:
    explicit failure(const string& msg, const error_code& ec = io_errc::stream);
    explicit failure(const char* msg, const error_code& ec = io_errc::stream);
  };  
}

1 An implementation is permitted to define ios_base::failure as a synonym for a class with equivalent functionality to class ios_base::failure shown in this subclause.

[Note 1: When ios_base::failure is a synonym for another type, that type is required to provide a nested type failure to emulate the injected-class-name. — end note]

The class failure defines the base class for the types of all objects thrown as exceptions, by functions in the iostreams library, to report errors detected during stream buffer operations.

2 When throwing ios_base::failure exceptions, implementations should provide values of ec that identify the specific reason for the failure.

[Note 2: Errors arising from the operating system would typically be reported as system_category() errors with an error value of the error number reported by the operating system. Errors arising from within the stream library would typically be reported as error_code(io_errc::stream, iostream_category()). — end note]

explicit failure(const string& msg, const error_code& ec = io_errc::stream);

3 Effects: Constructs the base class with msg and ec.

explicit failure(const char* msg, const error_code& ec = io_errc::stream);

4 Effects: Constructs the base class with msg and ec.

29.5.3.2.2 Type ios_base::fmtflags

using fmtflags = T1;

1 The type fmtflags is a bitmask type (16.3.3.3.4). Setting its elements has the effects indicated in Table 116.

2 Type fmtflags also defines the constants indicated in Table 117.

29.5.3.2.3 Type ios_base::iostate

using iostate = T2;

1 The type iostate is a bitmask type (16.3.3.3.4) that contains the elements indicated in Table 118.

2 Type iostate also defines the constant:

— goodbit, the value zero.

29.5.3.2.4 Type ios_base::openmode

using openmode = T3;

1 The type openmode is a bitmask type (16.3.3.3.4). It contains the elements indicated in Table 119.

29.5.3.2.5 Type ios_base::seekdir

using seekdir = T4;

1 The type seekdir is an enumerated type (16.3.3.3.3) that contains the elements indicated in Table 120.
Table 116: fmtflags effects  [tab:ios.fmtflags]

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolalpha</td>
<td>insert and extract bool type in alphabetic format</td>
</tr>
<tr>
<td>dec</td>
<td>converts integer input or generates integer output in decimal base</td>
</tr>
<tr>
<td>fixed</td>
<td>generate floating-point output in fixed-point notation</td>
</tr>
<tr>
<td>hex</td>
<td>converts integer input or generates integer output in hexadecimal base</td>
</tr>
<tr>
<td>internal</td>
<td>adds fill characters at a designated internal point in certain generated output, or identical to right if no such point is designated</td>
</tr>
<tr>
<td>left</td>
<td>adds fill characters on the right (final positions) of certain generated output</td>
</tr>
<tr>
<td>oct</td>
<td>converts integer input or generates integer output in octal base</td>
</tr>
<tr>
<td>right</td>
<td>adds fill characters on the left (initial positions) of certain generated output</td>
</tr>
<tr>
<td>scientific</td>
<td>generates floating-point output in scientific notation</td>
</tr>
<tr>
<td>showbase</td>
<td>generates a prefix indicating the numeric base of generated integer output</td>
</tr>
<tr>
<td>showpoint</td>
<td>generates a decimal-point character unconditionally in generated floating-point output</td>
</tr>
<tr>
<td>skipws</td>
<td>skips leading whitespace before certain input operations</td>
</tr>
<tr>
<td>unitbuf</td>
<td>flushes output after each output operation</td>
</tr>
<tr>
<td>uppercase</td>
<td>replaces certain lowercase letters with their uppercase equivalents in generated output</td>
</tr>
</tbody>
</table>

Table 117: fmtflags constants  [tab:ios.fmtflags.const]

<table>
<thead>
<tr>
<th>Constant</th>
<th>Allowable values</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustfield</td>
<td>left</td>
</tr>
<tr>
<td>basefield</td>
<td>dec</td>
</tr>
<tr>
<td>floatfield</td>
<td>scientific</td>
</tr>
</tbody>
</table>

Table 118: iostate effects  [tab:ios.iostate]

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>badbit</td>
<td>indicates a loss of integrity in an input or output sequence (such as an irrecoverable read error from a file);</td>
</tr>
<tr>
<td>eofbit</td>
<td>indicates that an input operation reached the end of an input sequence;</td>
</tr>
<tr>
<td>failbit</td>
<td>indicates that an input operation failed to read the expected characters, or that an output operation failed to generate the desired characters.</td>
</tr>
</tbody>
</table>

Table 119: openmode effects  [tab:ios.openmode]

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>seek to end before each write</td>
</tr>
<tr>
<td>ate</td>
<td>open and seek to end immediately after opening</td>
</tr>
<tr>
<td>binary</td>
<td>perform input and output in binary mode (as opposed to text mode)</td>
</tr>
<tr>
<td>in</td>
<td>open for input</td>
</tr>
<tr>
<td>out</td>
<td>open for output</td>
</tr>
<tr>
<td>trunc</td>
<td>truncate an existing stream when opening</td>
</tr>
</tbody>
</table>

Table 120: seekdir effects  [tab:ios.seekdir]

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>beg</td>
<td>request a seek (for subsequent input or output) relative to the beginning of the stream</td>
</tr>
<tr>
<td>cur</td>
<td>request a seek relative to the current position within the sequence</td>
</tr>
<tr>
<td>end</td>
<td>request a seek relative to the current end of the sequence</td>
</tr>
</tbody>
</table>
29.5.3.2.6 Class `ios_base::Init`  

```cpp
namespace std {
    class ios_base::Init {
    public:
        Init();
        Init(const Init&) = default;
        ~Init();
        Init& operator=(const Init&) = default;
    private:
        static int init_cnt;  // exposition only
    }
}
```

1. The class `Init` describes an object whose construction ensures the construction of the eight objects declared in `<iostream>` (29.4) that associate file stream buffers with the standard C streams provided for by the functions declared in `<cstdio>` (29.12.1).

2. For the sake of exposition, the maintained data is presented here as:

   — static int init_cnt, counts the number of constructor and destructor calls for class `Init`, initialized to zero.

   ```cpp
   Init();
   ```

3. **Effects**: Constructs and initializes the objects `cin`, `cout`, `cerr`, `clog`, `wcin`, `wcout`, `wcerr`, and `wclog` if they have not already been constructed and initialized.

   ```cpp
   ~Init();
   ```

4. **Effects**: If there are no other instances of the class still in existence, calls `cout.flush()`, `cerr.flush()`, `clog.flush()`, `wcout.flush()`, `wcerr.flush()`, and `wclog.flush()`.

29.5.3.3 State functions

```cpp
fmtflags flags() const;
```

1. **Returns**: The format control information for both input and output.

```cpp
fmtflags flags(fmtflags fmtfl);
```

2. **Postconditions**: `fmtfl == flags()`.

```cpp
fmtflags setf(fmtflags fmtfl);
```

4. **Effects**: Sets `fmtfl` in `flags()`.

```cpp
fmtflags setf(fmtflags fmtfl, fmtflags mask);
```

6. **Effects**: Clears `mask` in `flags()`, sets `fmtfl & mask` in `flags()`.

```cpp
void unsetf(fmtflags mask);
```

8. **Effects**: Clears `mask` in `flags()`.

```cpp
streamsize precision() const;
```

9. **Returns**: The precision to generate on certain output conversions.

```cpp
streamsize precision(streamsize prec);
```

10. **Postconditions**: `prec == precision()`.

```cpp
streamsize width() const;
```

12. **Returns**: The minimum field width (number of characters) to generate on certain output conversions.
streamsize width(streamsize wide);

13    Postconditions: wide == width().

14    Returns: The previous value of width().

29.5.3.4 Functions

locale imbue(const locale& loc);

1    Effects: Calls each registered callback pair (fn, idx) (29.5.3.7) as (*fn)(imbue_event, *this, idx)
at such a time that a call to ios_base::getloc() from within fn returns the new locale value loc.

2    Postconditions: loc == getloc().

3    Returns: The previous value of getloc().

locale getloc() const;

4    Returns: If no locale has been imbued, a copy of the global C++ locale, locale(), in effect at the time
          of construction. Otherwise, returns the imbued locale, to be used to perform locale-dependent input
          and output operations.

29.5.3.5 Static members

static bool sync_with_stdio(bool sync = true);

1    Effects: If any input or output operation has occurred using the standard streams prior to the call,
          the effect is implementation-defined. Otherwise, called with a false argument, it allows the standard
          streams to operate independently of the standard C streams.

2    Returns: true if the previous state of the standard iostream objects (29.4) was synchronized and
          otherwise returns false. The first time it is called, the function returns true.

3    Remarks: When a standard iostream object str is synchronized with a standard stdio stream f, the
          effect of inserting a character c by

          fputc(f, c);

          is the same as the effect of

          str.rdbuf()->sputc(c);

          for any sequences of characters; the effect of extracting a character c by

          c = fgetc(f);

          is the same as the effect of

          c = str.rdbuf()->sbumpc();

          for any sequences of characters; and the effect of pushing back a character c by

          ungetc(c, f);

          is the same as the effect of

          str.rdbuf()->sputbackc(c);

          for any sequence of characters.291

29.5.3.6 Storage functions

static int xalloc();

1    Returns: index ++.

2    Remarks: Concurrent access to this function by multiple threads does not result in a data race (6.9.2).

long& iword(int idx);

3    Preconditions: idx is a value obtained by a call to xalloc.

4    Effects: If iarray is a null pointer, allocates an array of long of unspecified size and stores a pointer
          to its first element in iarray. The function then extends the array pointed at by iarray as necessary

---

291) This implies that operations on a standard iostream object can be mixed arbitrarily with operations on the corresponding
stdio stream. In practical terms, synchronization usually means that a standard iostream object and a standard stdio object
share a buffer.
to include the element \(iarray[\text{idx}]\). Each newly allocated element of the array is initialized to zero. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to \code{copyfmt}, calling \code{iword} with the same index yields another reference to the same value. If the function fails and \code{*this} is a base class subobject of a \code{basic_ios<>} object or subobject, the effect is equivalent to calling \code{basic_ios<>::setState(badbit)} on the derived object (which may throw \code{failure}).

\begin{verbatim}
void* & pword(int idx);
\end{verbatim}

Returns: On success \code{iarray[idx]}. On failure, a valid \code{long&} initialized to 0.

### Callbacks

\begin{verbatim}
void register_callback(event_callback fn, int idx);
\end{verbatim}

Preconditions: \(\text{idx}\) is a value obtained by a call to \code{xalloc}.

Effects: If \code{parray} is a null pointer, allocates an array of pointers to \code{void} of unspecified size and stores a pointer to its first element in \code{parray}. The function then extends the array pointed at by \code{parray} as necessary to include the element \code{parray[\text{idx}]]. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to \code{copyfmt}, calling \code{pword} with the same index yields another reference to the same value. If the function fails and \code{*this} is a base class subobject of a \code{basic_ios<>} object or subobject, the effect is equivalent to calling \code{basic_ios<>::setState(badbit)} on the derived object (which may throw \code{failure}).

Returns: On success \code{parray[idx]}. On failure a valid \code{void* &} initialized to 0.

Remarks: After a subsequent call to \code{pword(int)} for the same object, the earlier return value may no longer be valid.

### Constructors and destructor

\begin{verbatim}
ios_base();
\end{verbatim}

Effects: Each \code{ios_base} member has an indeterminate value after construction. The object’s members shall be initialized by calling \code{basic_ios::init} before the object’s first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

\begin{verbatim}
~ios_base();
\end{verbatim}

Effects: Calls each registered callback pair \((\text{fn}, \text{idx})\) (29.5.3.7) as \((\text{fn})(\text{erase_event}, *\text{this}, \text{idx})\) at such time that any \code{ios_base} member function called from within \code{fn} has well-defined results.

### Class template fpos

\begin{verbatim}
namespace std {
    template<class stateT> class fpos {
    public:
        // 29.5.4.1, members
        stateT state() const;
        void state(stateT);
        // exposition only
    private:
        stateT st;
    }
}
\end{verbatim}

292) An implementation is free to implement both the integer array pointed at by \code{iarray} and the pointer array pointed at by \code{parray} as sparse data structures, possibly with a one-element cache for each.

293) For example, because it cannot allocate space.

294) For example, because it cannot allocate space.
29.5.4.1 Members

```cpp
void state(stateT s);
```

Effects: Assigns s to st.

```cpp
stateT state() const;
```

Returns: Current value of st.

29.5.4.2 Requirements

An fpos type specifies file position information. It holds a state object whose type is equal to the template parameter stateT. Type stateT shall meet the `Cpp17DefaultConstructible` (Table 27), `Cpp17CopyConstructible` (Table 29), `Cpp17CopyAssignable` (Table 31), and `Cpp17Destructible` (Table 32) requirements. If is_trivially_copy_constructible_v<stateT> is true, then fpos<stateT> has a trivial copy constructor. If is_trivially_copyAssignable_v<stateT> is true, then fpos<stateT> has a trivial copy assignment operator. If is_trivially_destructible_v<stateT> is true, then fpos<stateT> has a trivial destructor. All specializations of fpos meet the `Cpp17DefaultConstructible`, `Cpp17CopyConstructible`, `Cpp17CopyAssignable`, `Cpp17Destructible`, and `Cpp17EqualityComparable` (Table 25) requirements. In addition, the expressions shown in Table 121 are valid and have the indicated semantics. In that table,

1. P refers to an instance of fpos,
2. p and q refer to values of type P or const P,
3. pl and ql refer to modifiable lvalues of type P,
4. 0 refers to type streamoff, and
5. o refers to a value of type streamoff or const streamoff.

### Table 121: Position type requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(o)</td>
<td>P</td>
<td>converts from offset</td>
<td>Effects: Value-initializes the state object.</td>
</tr>
<tr>
<td>P p(o);</td>
<td>P</td>
<td>P(0)</td>
<td>Postconditions: p == P(o)</td>
</tr>
<tr>
<td>P p = o;</td>
<td>streamoff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0(p)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p != q</td>
<td>convertible to bool</td>
<td>! (p == q)</td>
<td>Remarks: With ql = p + o; then: ql - o == p</td>
</tr>
<tr>
<td>p + o</td>
<td>P</td>
<td>+ offset</td>
<td>Remarks: With ql = p1; before the +=, then: p1 - o == ql</td>
</tr>
<tr>
<td>pl += o</td>
<td>P&amp;</td>
<td>+= offset</td>
<td>Remarks: With ql = q1; before the +=, then: q1 + o == p</td>
</tr>
<tr>
<td>p - o</td>
<td>P</td>
<td>- offset</td>
<td>Remarks: With ql = q1; before the -=, then: p1 + o == q1</td>
</tr>
<tr>
<td>pl -= o</td>
<td>P&amp;</td>
<td>-= offset</td>
<td>Remarks: With ql = q1; before the -=, then: p1 + o == q1</td>
</tr>
<tr>
<td>o + p</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p - q</td>
<td>streamoff</td>
<td>distance</td>
<td></td>
</tr>
</tbody>
</table>

2 Stream operations that return a value of type traits::pos_type return P(0) as an invalid value to signal an error. If this value is used as an argument to any istream, ostream, or streambuf member that accepts a value of type traits::pos_type then the behavior of that function is undefined.

29.5.5 Class template basic_ios

29.5.5.1 Overview

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>>
```

§ 29.5.5.1
class basic_ios : public ios_base {
    public:
        using char_type = charT;
        using int_type = typename traits::int_type;
        using pos_type = typename traits::pos_type;
        using off_type = typename traits::off_type;
        using traits_type = traits;

        // 29.5.5.4, flags functions
        explicit operator bool() const;
        bool operator!() const;
        iostate rdstate() const;
        void clear(iostate state = goodbit);
        void setstate(iostate state);
        bool good() const;
        bool eof() const;
        bool fail() const;
        bool bad() const;

        iostate exceptions() const;
        void exceptions(iostate except);

        // 29.5.5.2, constructor/destructor
        explicit basic_ios(basic_streambuf<charT, traits>* sb);
        virtual ~basic_ios();

        // 29.5.5.3, members
        basic_ostream<charT, traits>* tie() const;
        basic_ostream<charT, traits>* tie(basic_ostream<charT, traits>* tiestr);

        basic_streambuf<charT, traits>* rdbuf() const;
        basic_streambuf<charT, traits>* rdbuf(basic_streambuf<charT, traits>* sb);

        basic_ios& copyfmt(const basic_ios& rhs);
        char_type fill() const;
        char_type fill(char_type ch);

        locale imbue(const locale& loc);

        char    narrow(char_type c, char dfault) const;
        char_type widen(char c) const;

        basic_ios(const basic_ios&) = delete;
        basic_ios& operator=(const basic_ios&) = delete;

    protected:
        basic_ios();
        void init(basic_streambuf<charT, traits>* sb);
        void move(basic_ios& rhs);
        void move(basic_ios&& rhs);
        void swap(basic_ios& rhs) noexcept;
        void set_rdbuf(basic_streambuf<charT, traits>* sb);
    }
};

29.5.5.2 Constructors

explicit basic_ios(basic_streambuf<charT, traits>* sb);

Effects: Assigns initial values to its member objects by calling init(sb).
basic_i os();

Effects: Leaves its member objects uninitialized. The object shall be initialized by calling basic_ios::init before its first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

~basic_i os();

Remarks: The destructor does not destroy rdbuf().

void init(basic_streambuf<CharT, Traits>* sb);

Postconditions: The postconditions of this function are indicated in Table 122.

Table 122: basic_i os::init() effects [tab:basic.i os.cons]

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdbuf()</td>
<td>sb</td>
</tr>
<tr>
<td>tie()</td>
<td>0</td>
</tr>
<tr>
<td>rdstate()</td>
<td>goodbit if sb is not a null pointer, otherwise badbit.</td>
</tr>
<tr>
<td>exceptions()</td>
<td>goodbit</td>
</tr>
<tr>
<td>flags()</td>
<td>skipws</td>
</tr>
<tr>
<td>width()</td>
<td>0</td>
</tr>
<tr>
<td>precision()</td>
<td>6</td>
</tr>
<tr>
<td>fill()</td>
<td>widen(‘ ’)</td>
</tr>
<tr>
<td>getloc()</td>
<td>a copy of the value returned by locale()</td>
</tr>
<tr>
<td>iarray</td>
<td>a null pointer</td>
</tr>
<tr>
<td>parray</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

29.5.5.3 Member functions [basic.i os.members]

basic_ostream<CharT, Traits>* tie() const;

Returns: An output sequence that is tied to (synchronized with) the sequence controlled by the stream buffer.

basic_ostream<CharT, Traits>* tie(basic_ostream<CharT, Traits>* tiestr);

Preconditions: If tiestr is not null, tiestr is not reachable by traversing the linked list of tied stream objects starting from tiestr->tie().

Postconditions: tiestr == tie().

Returns: The previous value of tie().

basic_streambuf<CharT, Traits>* rdbuf() const;

Returns: A pointer to the streambuf associated with the stream.

basic_streambuf<CharT, Traits>* rdbuf(basic_streambuf<CharT, Traits>* sb);

Effects: Calls clear().

Postconditions: sb == rdbuf().

Returns: The previous value of rdbuf().

locale imbue(const locale& loc);

Effects: Calls ios_base::imbue(loc) (29.5.3.4) and if rdbuf() != 0 then rdbuf()->pubimbue(loc) (29.6.3.3.1).

Returns: The prior value of ios_base::imbue().

char narrow(char_type c, char dfault) const;

Returns: use_facet<ctype<char_type>>(getloc()).narrow(c, dfault)
char_type widen(char c) const;

    Returns: use_facet<ctype<char_type>>().widen(c)

char_type fill() const;

    Returns: The character used to pad (fill) an output conversion to the specified field width.

char_type fill(char_type fillch);

    Postconditions: traits::eq(fillch, fill()).

    Returns: The previous value of fill().

basic_ios& copyfmt(const basic_ios& rhs);

    Effects: If (this == addressof(rhs)) is true does nothing. Otherwise assigns to the member objects of *this the corresponding member objects of rhs as follows:

    — calls each registered callback pair (fn, idx) as (*fn)(erase_event, *this, idx);
(16.1)

    — then, assigns to the member objects of *this the corresponding member objects of rhs, except that
(16.2)

        — rdstate(), rdbuf(), and exceptions() are left unchanged;
(16.2.1)

        — the contents of arrays pointed at by pword and iword are copied, not the pointers themselves;\(^{295}\) and
(16.2.2)

        — if any newly stored pointer values in *this point at objects stored outside the object rhs and those objects are destroyed when rhs is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects;
(16.2.3)

    — then, calls each callback pair that was copied from rhs as (*fn)(copyfmt_event, *this, idx);
(16.3)

    — then, calls exceptions(rhs.exceptions()).
(16.4)

    [Note 1: The second pass through the callback pairs permits a copied pword value to be zeroed, or to have its referent deep copied or reference counted, or to have other special action taken. — end note]

    Postconditions: The postconditions of this function are indicated in Table 123.

Table 123: basic_ios::copyfmt() effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdbuf()</td>
<td>unchanged</td>
</tr>
<tr>
<td>tie()</td>
<td>rhs.tie()</td>
</tr>
<tr>
<td>rdstate()</td>
<td>unchanged</td>
</tr>
<tr>
<td>exceptions()</td>
<td>rhs.exceptions()</td>
</tr>
<tr>
<td>flags()</td>
<td>rhs.flags()</td>
</tr>
<tr>
<td>width()</td>
<td>rhs.width()</td>
</tr>
<tr>
<td>precision()</td>
<td>rhs.precision()</td>
</tr>
<tr>
<td>fill()</td>
<td>rhs.fill()</td>
</tr>
<tr>
<td>getloc()</td>
<td>rhs.getloc()</td>
</tr>
</tbody>
</table>

    Returns: *this.

void move(basic_ios& rhs);
void move(basic_ios&& rhs);

    Postconditions: *this has the state that rhs had before the function call, except that rdbuf() returns nullptr. rhs is in a valid but unspecified state, except that rhs.rdbuf() returns the same value as it returned before the function call, and rhs.tie() returns nullptr.

\(^{295}\) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is nonzero.
void swap(basic_ios& rhs) noexcept;

Effects: The states of *this and rhs are exchanged, except that rdbuf() returns the same value as it returned before the function call, and rhs.rdbuf() returns the same value as it returned before the function call.

void set_rdbuf(basic_streambuf<charT, traits>* sb);

Preconditions: sb != nullptr is true.

Effects: Associates the basic_streambuf object pointed to by sb with this stream without calling clear().

Postconditions: rdbuf() == sb is true.

Throws: Nothing.

29.5.5.4 Flags functions

explicit operator bool() const;

Returns: !fail().

bool operator!() const;

Returns: fail().

iostate rdstate() const;

Returns: The error state of the stream buffer.

void clear(iostate state = goodbit);

Effects: If ((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0, returns. Otherwise, the function throws an object of class ios_base::failure (29.5.3.2.1), constructed with implementation-defined argument values.

Postconditions: If rdbuf() != 0 then state == rdstate(); otherwise rdstate() == (state | ios_base::badbit).

void setstate(iostate state);

Effects: Calls clear(rdstate() | state) (which may throw ios_base::failure (29.5.3.2.1)).

bool good() const;

Returns: rdstate() == 0

bool eof() const;

Returns: true if eofbit is set in rdstate().

bool fail() const;

Returns: true if failbit or badbit is set in rdstate().

bool bad() const;

Returns: true if badbit is set in rdstate().

iostate exceptions() const;

Returns: A mask that determines what elements set in rdstate() cause exceptions to be thrown.

void exceptions(iostate except);

Effects: Calls clear(rdstate()).

Postconditions: except == exceptions().

29.5.6 ios_base manipulators

29.5.6.1 fmtflags manipulators

Each function specified in this subclause is a designated addressable function (16.4.5.2.1).

Checking badbit also for fail() is historical practice.
ios_base& boolalpha(ios_base& str);
   Effects: Calls str.setf(ios_base::boolalpha).
   Returns: str.

ios_base& noboolalpha(ios_base& str);
   Effects: Calls str.unsetf(ios_base::boolalpha).
   Returns: str.

ios_base& showbase(ios_base& str);
   Effects: Calls str.setf(ios_base::showbase).
   Returns: str.

ios_base& noshowbase(ios_base& str);
   Effects: Calls str.unsetf(ios_base::showbase).
   Returns: str.

ios_base& showpoint(ios_base& str);
   Effects: Calls str.setf(ios_base::showpoint).
   Returns: str.

ios_base& noshowpoint(ios_base& str);
   Effects: Calls str.unsetf(ios_base::showpoint).
   Returns: str.

ios_base& showpos(ios_base& str);
   Effects: Calls str.setf(ios_base::showpos).
   Returns: str.

ios_base& noshowpos(ios_base& str);
   Effects: Calls str.unsetf(ios_base::showpos).
   Returns: str.

ios_base& skipws(ios_base& str);
   Effects: Calls str.setf(ios_base::skipws).
   Returns: str.

ios_base& noskipws(ios_base& str);
   Effects: Calls str.unsetf(ios_base::skipws).
   Returns: str.

ios_base& uppercase(ios_base& str);
   Effects: Calls str.setf(ios_base::uppercase).
   Returns: str.

ios_base& nouppercase(ios_base& str);
   Effects: Calls str.unsetf(ios_base::uppercase).
   Returns: str.

ios_base& unitbuf(ios_base& str);
   Effects: Calls str.setf(ios_base::unitbuf).
   Returns: str.

ios_base& nounitbuf(ios_base& str);
   Effects: Calls str.unsetf(ios_base::unitbuf).
Returns: \texttt{str}.

### 29.5.6.2 \textbf{adjustfield manipulators} \hfill [\textit{adjustfield.manip}]

Each function specified in this subclause is a designated addressable function (16.4.5.2.1).

- \texttt{ios\_base\& \text{internal}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::internal, ios\_base::adjustfield)}.
  \textit{Returns:} \texttt{str}.

- \texttt{ios\_base\& \text{left}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::left, ios\_base::adjustfield)}.
  \textit{Returns:} \texttt{str}.

- \texttt{ios\_base\& \text{right}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::right, ios\_base::adjustfield)}.
  \textit{Returns:} \texttt{str}.

### 29.5.6.3 \textbf{basefield manipulators} \hfill [\textit{basefield.manip}]

Each function specified in this subclause is a designated addressable function (16.4.5.2.1).

- \texttt{ios\_base\& \text{dec}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::dec, ios\_base::basefield)}.
  \textit{Returns:} \texttt{str}.

- \texttt{ios\_base\& \text{hex}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::hex, ios\_base::basefield)}.
  \textit{Returns:} \texttt{str}.

- \texttt{ios\_base\& \text{oct}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::oct, ios\_base::basefield)}.
  \textit{Returns:} \texttt{str}.

### 29.5.6.4 \textbf{floatfield manipulators} \hfill [\textit{floatfield.manip}]

Each function specified in this subclause is a designated addressable function (16.4.5.2.1).

- \texttt{ios\_base\& \text{fixed}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::fixed, ios\_base::floatfield)}.
  \textit{Returns:} \texttt{str}.

- \texttt{ios\_base\& \text{scientific}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::scientific, ios\_base::floatfield)}.
  \textit{Returns:} \texttt{str}.

- \texttt{ios\_base\& \text{hexfloat}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::fixed | ios\_base::scientific, ios\_base::floatfield)}.
  \textit{Returns:} \texttt{str}.

\[\text{Note 1:} \text{The more obvious use of} \ \text{ios\_base::hex} \ \text{to specify hexadecimal floating-point format would change the meaning of existing well-defined programs. C++ 2003 gives no meaning to the combination of} \ \text{fixed} \ \text{and} \ \text{scientific}. \ - \ \text{end note}\]

- \texttt{ios\_base\& \text{defaultfloat}(ios\_base\& \text{str});}
  \textit{Effects:} Calls \texttt{str.unsetf(ios\_base::floatfield)}.
  \textit{Returns:} \texttt{str}.

297) The function signature \texttt{dec(ios\_base\&)} can be called by the function signature \texttt{basic\_ostream\& stream::operator\ll(ios\_base\& \text{str})} to permit expressions of the form \texttt{cout \ll \text{dec}} to change the format flags stored in \texttt{cout}.

\textsection{29.5.6.4}
29.5.7 Error reporting

```cpp
error_code make_error_code(io_errc e) noexcept;
1 Returns: error_code(static_cast<int>(e), iostream_category()).
```

```cpp
error_condition make_error_condition(io_errc e) noexcept;
2 Returns: error_condition(static_cast<int>(e), iostream_category()).
```

```cpp
const error_category& iostream_category() noexcept;
3 Returns: A reference to an object of a type derived from class error_category.
4 The object’s default_error_condition and equivalent virtual functions shall behave as specified
   for the class error_category. The object’s name virtual function shall return a pointer to the string
   "iostream".
```

29.6 Stream buffers

29.6.1 Header <streambuf> synopsis

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT> >
    class basic_streambuf;
    using streambuf = basic_streambuf<char> ;
    using wstreambuf = basic_streambuf<wchar_t> ;
}
```

1 The header <streambuf> defines types that control input from and output to character sequences.

29.6.2 Stream buffer requirements

1 Stream buffers can impose various constraints on the sequences they control. Some constraints are:

(1.1) — The controlled input sequence can be not readable.
(1.2) — The controlled output sequence can be not writable.
(1.3) — The controlled sequences can be associated with the contents of other representations for character
   sequences, such as external files.
(1.4) — The controlled sequences can support operations directly to or from associated sequences.
(1.5) — The controlled sequences can impose limitations on how the program can read characters from a
   sequence, write characters to a sequence, put characters back into an input sequence, or alter the stream
   position.

2 Each sequence is characterized by three pointers which, if non-null, all point into the same charT array object.
   The array object represents, at any moment, a (sub)sequence of characters from the sequence. Operations
   performed on a sequence alter the values stored in these pointers, perform reads and writes directly to or
   from associated sequences, and alter “the stream position” and conversion state as needed to maintain this
   subsequence relationship. The three pointers are:

(2.1) — the beginning pointer, or lowest element address in the array (called xbeg here);
(2.2) — the next pointer, or next element address that is a current candidate for reading or writing (called
       xnext here);
(2.3) — the end pointer, or first element address beyond the end of the array (called xend here).

3 The following semantic constraints shall always apply for any set of three pointers for a sequence, using the
   pointer names given immediately above:

(3.1) — If xnext is not a null pointer, then xbeg and xend shall also be non-null pointers into the same charT
   array, as described above; otherwise, xbeg and xend shall also be null.
(3.2) — If xnext is not a null pointer and xnext < xend for an output sequence, then a write position is
   available. In this case, *xnext shall be assignable as the next element to write (to put, or to store a
   character value, into the sequence).
(3.3) — If xnext is not a null pointer and xbeg < xnext for an input sequence, then a putback position is
   available. In this case, xnext[-1] shall have a defined value and is the next (preceding) element to
   store a character that is put back into the input sequence.
If \( x_{\text{next}} \) is not a null pointer and \( x_{\text{next}} < x_{\text{end}} \) for an input sequence, then a \textit{read position} is available. In this case, \( *x_{\text{next}} \) shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

29.6.3 Class template \texttt{basic\_streambuf}

29.6.3.1 General

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_streambuf {
        public:
            using char_type = charT;
            using int_type = typename traits::int_type;
            using pos_type = typename traits::pos_type;
            using off_type = typename traits::off_type;
            using traits_type = traits;
            virtual ~basic_streambuf();

            // 29.6.3.3.1, locales
            locale pubimbue(const locale& loc);
            locale getloc() const;

            // 29.6.3.3.2, buffer and positioning
            basic_streambuf* pubsetbuf(char_type* s, streamsize n);
            pos_type pubseekoff(off_type off, ios_base::seekdir way,
                                 ios_base::openmode which = ios_base::in | ios_base::out);
            pos_type pubseekpos(pos_type sp,
                                 ios_base::openmode which = ios_base::in | ios_base::out);
            int pubsync();

            // get and put areas
            // 29.6.3.3.3, get area
            streamsize in_avail();
            int_type snextc();
            int_type sbumpc();
            int_type sgetc();
            streamsize sgetn(char_type* s, streamsize n);

            // 29.6.3.3.4, putback
            int_type sputbackc(char_type c);
            int_type sungetc();

            // 29.6.3.3.5, put area
            int_type sputc(char_type c);
            streamsize sputn(const char_type* s, streamsize n);

        protected:
            basic_streambuf();
            basic_streambuf(const basic_streambuf& rhs);
            basic_streambuf& operator=(const basic_streambuf& rhs);
            void swap(basic_streambuf& rhs);

            // 29.6.3.4.2, get area access
            char_type* eback() const;
            char_type* gptr() const;
            char_type* egptr() const;
            void gbump(int n);
            void setg(char_type* gbeg, char_type* gnext, char_type* gend);
        }
```
The class template `basic_streambuf` serves as an abstract base class for deriving various stream buffers whose objects each control two character sequences:

- (1.1) a character input sequence;
- (1.2) a character output sequence.

### 29.6.3.2 Constructors

```cpp
basic_streambuf();
```

**Effects**: Initializes:

1. (1.1) all pointer member objects to null pointers,
2. (1.2) the `getloc()` member to a copy the global locale, `locale()`, at the time of construction.

**Remarks**: Once the `getloc()` member is initialized, results of calling locale member functions, and of members of facets so obtained, can safely be cached until the next time the member `imbue` is called.

```cpp
basic_streambuf(const basic_streambuf& rhs);
```

**Postconditions**:

1. (3.1) `eback()` == `rhs.eback()`
2. (3.2) `gptr()` == `rhs.gptr()`
3. (3.3) `egptr()` == `rhs.egptr()`
4. (3.4) `pbase()` == `rhs.pbase()`

---

298) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class can be constructed.
(3.5) \[ \text{pptr()} == \text{rhs.pptr()} \]

(3.6) \[ \text{epptr()} == \text{rhs.epptr()} \]

(3.7) \[ \text{getloc()} == \text{rhs.getloc()} \]

~basic_streambuf();

Effects: None.

29.6.3.3 Public member functions

29.6.3.3.1 Locales

locale pubimbue(const locale& loc);

Effects: Calls imbue(loc).
Postconditions: loc == getloc().
Returns: Previous value of getloc().

locale getloc() const;
Returns: If pubimbue() has ever been called, then the last value of loc supplied, otherwise the current
global locale, locale(), in effect at the time of construction. If called after pubimbue() has been called
but before pubimbue has returned (i.e., from within the call of imbue()) then it returns the previous
value.

29.6.3.3.2 Buffer management and positioning

basic_streambuf* pubsetbuf(char_type* s, streamsize n);
Returns: setbuf(s, n).

pos_type pubseekoff(off_type off, ios_base::seekdir way,
ios_base::openmode which
= ios_base::in | ios_base::out);
Returns: seekoff(off, way, which).

pos_type pubseekpos(pos_type sp,
ios_base::openmode which
= ios_base::in | ios_base::out);
Returns: seekpos(sp, which).

int pubsync();
Returns: sync().

29.6.3.3.3 Get area

streamsize in_avail();
Returns: If a read position is available, returns egptr() - gptr(). Otherwise returns showmanyc()
(29.6.3.5.3).

int_type snextc();
Effects: Calls sbumpc().
Returns: If that function returns traits::eof(), returns traits::eof(). Otherwise, returns sgetc().

int_type sbumpc();
Effects: If the input sequence read position is not available, returns uflow(). Otherwise, returns
traits::to_int_type(*gptr()) and increments the next pointer for the input sequence.

int_type sgetc();
Returns: If the input sequence read position is not available, returns underflow(). Otherwise, returns
traits::to_int_type(*gptr()).
streamsize sgetn(char_type* s, streamsize n);

Returns: xsgetn(s, n).

29.6.3.3.4 Putback

int_type sputbackc(char_type c);

Effects: If the input sequence putback position is not available, or if traits::eq(c, gptr()[-1]) is false, returns pbackfail(traits::to_int_type(c)). Otherwise, decrements the next pointer for the input sequence and returns traits::to_int_type(*gptr()).

int_type sungetc();

Effects: If the input sequence putback position is not available, returns pbackfail(). Otherwise, decrements the next pointer for the input sequence and returns traits::to_int_type(*gptr()).

29.6.3.3.5 Put area

int_type sputc(char_type c);

Effects: If the output sequence write position is not available, returns overflow(traits::to_int_type(c)). Otherwise, stores c at the next pointer for the output sequence, increments the pointer, and returns traits::to_int_type(c).

streamsize sputn(const char_type* s, streamsize n);

Returns: xsputn(s, n).

29.6.3.4 Protected member functions

29.6.3.4.1 Assignment

basic_streambuf& operator=(const basic_streambuf& rhs);

Postconditions:
(1.1)  eback() == rhs.eback()
(1.2)  gptr() == rhs.gptr()
(1.3)  egptr() == rhs.egptr()
(1.4)  pbase() == rhs.pbase()
(1.5)  pptr() == rhs.pptr()
(1.6)  eptr() == rhs.eptr()
(1.7)  getloc() == rhs.getloc()

Returns: *this.

void swap(basic_streambuf& rhs);

Effects: Swaps the data members of rhs and *this.

29.6.3.4.2 Get area access

char_type* eback() const;

Returns: The beginning pointer for the input sequence.

cchar_type* gptr() const;

Returns: The next pointer for the input sequence.

cchar_type* egptr() const;

Returns: The end pointer for the input sequence.

void gbump(int n);

Effects: Adds n to the next pointer for the input sequence.

void setg(char_type* gbeg, char_type* gnext, char_type* gend);

Postconditions: gbeg == eback(), gnext == gptr(), and gend == egptr() are all true.
29.6.3.4.3 Put area access

char_type* pbase() const;
1
   Returns: The beginning pointer for the output sequence.

char_type* pptr() const;
2
   Returns: The next pointer for the output sequence.

char_type* epptr() const;
3
   Returns: The end pointer for the output sequence.

void pbump(int n);
4
   Effects: Adds \( n \) to the next pointer for the output sequence.

void setp(char_type* pbeg, char_type* pend);
5
   Postconditions: \( \text{pbeg} == \text{pbase}() \), \( \text{pbeg} == \text{pptr}() \), and \( \text{pend} == \text{epptr}() \) are all true.

29.6.3.5 Virtual functions

29.6.3.5.1 Locales

void imbue(const locale&);
1
   Effects: Change any translations based on locale.

   Remarks: Allows the derived class to be informed of changes in locale at the time they occur. Between
   invocations of this function a class derived from streambuf can safely cache results of calls to locale
   functions and to members of facets so obtained.

   Default behavior: Does nothing.

29.6.3.5.2 Buffer management and positioning

basic_streambuf* setbuf(char_type* s, streamsize n);
1
   Effects: Influences stream buffering in a way that is defined separately for each class derived from
   basic_streambuf in this Clause (29.8.2.5, 29.9.2.5).

   Default behavior: Does nothing. Returns this.

pos_type seekoff(off_type off, ios_base::seekdir way,
   ios_base::openmode which = ios_base::in | ios_base::out);
3
   Effects: Alters the stream positions within one or more of the controlled sequences in a way that is
   defined separately for each class derived from basic_streambuf in this Clause (29.8.2.5, 29.9.2.5).

   Default behavior: Returns pos_type(off_type(-1)).

pos_type seekpos(pos_type sp,
   ios_base::openmode which = ios_base::in | ios_base::out);
5
   Effects: Alters the stream positions within one or more of the controlled sequences in a way that is
   defined separately for each class derived from basic_streambuf in this Clause (29.8.2, 29.9.2).

   Default behavior: Returns pos_type(off_type(-1)).

int sync();
7
   Effects: Synchronizes the controlled sequences with the arrays. That is, if \( \text{pbase}() \) is non-null the
   characters between \( \text{pbase}() \) and \( \text{pptr}() \) are written to the controlled sequence. The pointers may then
   be reset as appropriate.

   Returns: -1 on failure. What constitutes failure is determined by each derived class (29.9.2.5).

   Default behavior: Returns zero.
29.6.3.5.3 Get area

streamsize showmanyc();  \( ^{299} \)

Returns: An estimate of the number of characters available in the sequence, or -1. If it returns a positive value, then successive calls to underflow() will not return traits::eof() until at least that number of characters have been extracted from the stream. If showmanyc() returns -1, then calls to underflow() or uflow() will fail.  \( ^{300} \)

Default behavior: Returns zero.

Remarks: Uses traits::eof().

streamsize xsgetn(char_type* s, streamsize n);  \( ^{301} \)

Effects: Assigns up to n characters to successive elements of the array whose first element is designated by s. The characters assigned are read from the input sequence as if by repeated calls to sbumpc(). Assigning stops when either n characters have been assigned or a call to sbumpc() would return traits::eof().

Returns: The number of characters assigned.

Remarks: Uses traits::eof().

int_type underflow();  \( ^{301} \)

Returns: traits::to_int_type(c), where c is the first character of the pending sequence, without moving the input sequence position past it. If the pending sequence is null then the function returns traits::eof() to indicate failure.

Remarks: The public members of basic_streambuf call this virtual function only if gptr() is null or gptr() >= egptr().

The pending sequence of characters is defined as the concatenation of

- the empty sequence if gptr() is null, otherwise the characters in [gptr(), egptr()), followed by

- some (possibly empty) sequence of characters read from the input sequence.

The result character is the first character of the pending sequence if it is non-empty, otherwise the next character that would be read from the input sequence.

The backup sequence is the empty sequence if eback() is null, otherwise the characters in [eback(), gptr()).

Effects: The function sets up the gptr() and egptr() such that if the pending sequence is non-empty, then egptr() is non-null and the characters in [gptr(), egptr()) are the characters in the pending sequence, otherwise either gptr() is null or gptr() == egptr().

If eback() and gptr() are non-null then the function is not constrained as to their contents, but the usual backup condition” is that either

- the backup sequence contains at least gptr() - eback() characters, in which case the characters in [eback(), gptr()) agree with the last gptr() - eback() characters of the backup sequence, or

- the characters in [gptr() - n, gptr()) agree with the backup sequence (where n is the length of the backup sequence).

Default behavior: Returns traits::eof().

int_type uflow();

Preconditions: The constraints are the same as for underflow(), except that the result character is transferred from the pending sequence to the backup sequence, and the pending sequence is not empty before the transfer.

---

299) The morphemes of showmanyc are “es-how-many-see”, not “show-manic”.
300) underflow or uflow might fail by throwing an exception prematurely. The intention is not only that the calls will not return eof() but that they will return “immediately”.
301) Classes derived from basic_streambuf can provide more efficient ways to implement xsgetn() and xsputn() by overriding these definitions from the base class.
Default behavior: Calls `underflow()`. If `underflow()` returns `traits::eof()`, returns `traits::eof()`. Otherwise, returns the value of `traits::to_int_type(*gptr())` and increment the value of the next pointer for the input sequence.

Returns: `traits::eof()` to indicate failure.

### 29.6.3.5.4 Putback

```cpp
default behavior

```int_type pbackfail(int_type c = traits::eof());

```Remarks: The public functions of `basic_streambuf` call this virtual function only when `gptr()` is null, `gptr() == eback()`, or `traits::eq(traits::to_char_type(c), gptr()[-1])` returns `false`. Other calls shall also satisfy that constraint.

The **pending sequence** is defined as for `underflow()`, with the modifications that

1. If `traits::eq_int_type(c, traits::eof())` returns `true`, then the input sequence is backed up one character before the pending sequence is determined.
2. If `traits::eq_int_type(c, traits::eof())` returns `false`, then `c` is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

Returns: `traits::eof()` to indicate failure. Failure may occur because the input sequence could not be backed up, or if for some other reason the pointers could not be set consistent with the constraints. `pbackfail()` is called only when put back has really failed.

Returns some value other than `traits::eof()` to indicate success.

Default behavior: Returns `traits::eof()`.

### 29.6.3.5.5 Put area

```cpp
streamsize xsputn(const char_type* s, streamsize n);

```Effects: Writes up to `n` characters to the output sequence as if by repeated calls to `sputc(c)`. The characters written are obtained from successive elements of the array whose first element is designated by `s`. Writing stops when either `n` characters have been written or a call to `sputc(c)` would return `traits::eof()`. It is unspecified whether the function calls `overflow()` when `pptr() == epptr()` becomes `true` or whether it achieves the same effects by other means.

Returns: The number of characters written.

```cpp
int_type overflow(int_type c = traits::eof());

```Effects: Consumes some initial subsequence of the characters of the **pending sequence**. The pending sequence is defined as the concatenation of

1. The empty sequence if `pbase()` is null, otherwise the `pptr() - pbase()` characters beginning at `pbase()`, followed by
2. The empty sequence if `traits::eq_int_type(c, traits::eof())` returns `true`, otherwise the sequence consisting of `c`.

Remarks: The member functions `sputc()` and `sputn()` call this function in case that no room can be found in the put buffer enough to accommodate the argument character sequence.

Preconditions: Every overriding definition of this virtual function obeys the following constraints:

1. The effect of consuming a character on the associated output sequence is specified.\(^{302}\)
2. Let `r` be the number of characters in the pending sequence not consumed. If `r` is nonzero then `pbase()` and `pptr()` are set so that: `pptr() - pbase() == r` and the `r` characters starting at `pbase()` are the associated output stream. In case `r` is zero (all characters of the pending sequence have been consumed) then either `pbase()` is set to `nullptr`, or `pbase()` and `pptr()` are both set to the same non-null value.

---

\(^{302}\) That is, for each class derived from an instance of `basic_streambuf` in this Clause (29.8.2, 29.9.2), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.
The function may fail if either appending some character to the associated output stream fails or if it is unable to establish `pbase()` and `pptr()` according to the above rules.

*Returns:* `traits::eof()` or throws an exception if the function fails. Otherwise, returns some value other than `traits::eof()` to indicate success.

*Default behavior:* Returns `traits::eof()`.

### 29.7 Formatting and manipulators

#### 29.7.1 Header `<istream>` synopsis

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_istream;

    using istream = basic_istream<char>;
    using wistream = basic_istream<wchar_t>;

    template<class charT, class traits = char_traits<charT>>
    class basic_iostream;

    using iostream = basic_iostream<char>;
    using wiostream = basic_iostream<wchar_t>;

    template<class charT, class traits = char_traits<charT>>
    basic_istream<charT, traits>& ws(basic_istream<charT, traits>& is);

    template<class Istream, class T>
    Istream&& operator>>(Istream&& is, T&& x);
}
```

#### 29.7.2 Header `<ostream>` synopsis

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_ostream;

    using ostream = basic_ostream<char>;
    using wostream = basic_ostream<wchar_t>;

    template<class charT, class traits = char_traits<charT>>
    basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os);

    template<class charT, class traits = char_traits<charT>>
    basic_ostream<charT, traits>& ends(basic_ostream<charT, traits>& os);

    template<class charT, class traits = char_traits<charT>>
    basic_ostream<charT, traits>& flush(basic_ostream<charT, traits>& os);

    template<class charT, class traits = char_traits<charT>>
    basic_ostream<charT, traits>& emit_on_flush(basic_ostream<charT, traits>& os);

    template<class charT, class traits = char_traits<charT>>
    basic_ostream<charT, traits>& noemit_on_flush(basic_ostream<charT, traits>& os);

    template<class charT, class traits = char_traits<charT>>
    basic_ostream<charT, traits>& flush_emit(basic_ostream<charT, traits>& os);

    template<class Ostream, class T>
    Ostream&& operator<<(Ostream&& os, const T& x);
}
```

#### 29.7.3 Header `<iomanip>` synopsis

```cpp
namespace std {
    unspecified resetiosflags(ios_base::fmtflags mask);
    unspecified setiosflags (ios_base::fmtflags mask);
}
```

---

303) Typically, `overflow` returns `c` to indicate success, except when `traits::eq_int_type(c, traits::eof())` returns `true`, in which case it returns `traits::not_eof(c)`. 
template<class charT> unspecified setbase(int base);
template<class charT> unspecified setfill(charT c);
unspecified setprecision(int n);
unspecified setw(int n);
template<class moneyT> unspecified get_money(moneyT& mon, bool intl = false);
template<class moneyT> unspecified put_money(const moneyT& mon, bool intl = false);
template<class charT> unspecified get_time(struct tm* tmb, const charT* fmt);
template<class charT> unspecified put_time(const struct tm* tmb, const charT* fmt);

template<class charT>
unspecified quoted(const charT* s, charT delim = charT(‘\’)), charT escape = charT(‘\’));

template<class charT, class traits, class Allocator>
unspecified quoted(const basic_string< Traits, Allocator>& s,
charT delim = charT(‘\’)), charT escape = charT(‘\’));

template<class charT, class traits, class Allocator>
unspecified quoted(basic_string_view< charT, Traits > s,
charT delim = charT(‘\’)), charT escape = charT(‘\’));

29.7.4 Input streams
29.7.4.1 General

The header `<istream>` defines two types and a function signature that control input from a stream buffer along with a function template that extracts from stream rvalues.

29.7.4.2 Class template basic_istream

namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_istream : virtual public basic_ios<charT, traits> {
        public:
            // types (inherited from basic_ios (29.5.5))
            using char_type = charT;
            using int_type = typename traits::int_type;
            using pos_type = typename traits::pos_type;
            using off_type = typename traits::off_type;
            using traits_type = traits;

            // 29.7.4.2.2, constructor/destructor
            explicit basic_istream(basic_streambuf<charT, traits>* sb);
            virtual ~basic_istream();

            // 29.7.4.2.4, prefix/suffix
            class sentry;

            // 29.7.4.3, formatted input
            basic_istream& operator>>(basic_istream& (*pf)(basic_istream&));
            basic_istream& operator>>(basic_ios< charT, traits >& (*pf)(basic_ios< charT, traits >&));
            basic_istream& operator>>(ios_base& (*pf)(ios_base&));

            basic_istream& operator>>(bool& n);
            basic_istream& operator>>(short& n);
            basic_istream& operator>>(unsigned short& n);
            basic_istream& operator>>(int& n);
basic_istream<char, traits>& operator>>(unsigned int& n);
basic_istream<char, traits>& operator>>(long& n);
basic_istream<char, traits>& operator>>(unsigned long& n);
basic_istream<char, traits>& operator>>(long long& n);
basic_istream<char, traits>& operator>>(unsigned long long& n);
basic_istream<char, traits>& operator>>(float& f);
basic_istream<char, traits>& operator>>(double& f);
basic_istream<char, traits>& operator>>(void*& p);
basic_istream<char, traits>& operator>>(basic_streambuf<char_type, traits>* sb);

// 29.7.4.4, unformatted input
streamsize gcount() const;
int_type get();
basic_istream<char, traits>& get(char_type& c);
basic_istream<char, traits>& get(char_type* s, streamsize n);
basic_istream<char, traits>& get(char_type* s, streamsize n, char_type delim);
basic_istream<char, traits>& get(basic_streambuf<char_type, traits>* sb);
basic_istream<char, traits>& get(basic_streambuf<char_type, traits>* sb, char_type delim);

basic_istream<char, traits>& getline(char_type* s, streamsize n);
basic_istream<char, traits>& getline(char_type* s, streamsize n, char_type delim);
basic_istream<char, traits>& ignore(streamsize n = 1, int_type delim = traits::eof());
int_type peek();
basic_istream<char, traits>& read(char_type* s, streamsize n);
streamsize readsome(char_type* s, streamsize n);

basic_istream<char, traits>& putback(char_type c);
basic_istream<char, traits>& unget();
int sync();
pos_type tellg();
basic_istream<char, traits>& seekg(pos_type);
basic_istream<char, traits>& seekg(off_type, ios_base::seekdir);

protected:
// 29.7.4.2.2, copy/move constructor
basic_istream(const basic_istream&) = delete;
basic_istream(basic_istream&& rhs);

// 29.7.4.2.3, assign and swap
basic_istream& operator=(const basic_istream&) = delete;
basic_istream& operator=(basic_istream&& rhs);
void swap(basic_istream& rhs);

};

// 29.7.4.3.3, character extraction templates
template<class charT, class traits>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&, charT&);
template<class traits>
basic_istream<char, traits>& operator>>(basic_istream<char, traits>&, unsigned char&);
template<class traits>
basic_istream<char, traits>& operator>>(basic_istream<char, traits>&, signed char&);
template<class charT, class traits, size_t N>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&, charT(N));
template<class traits, size_t N>
basic_istream<char, traits>& operator>>(basic_istream<char, traits>&, unsigned char(N));
template<class traits, size_t N>
basic_istream<char, traits>& operator>>(basic_istream<char, traits>&, signed char(N));
The class template basic_istream defines a number of member function signatures that assist in reading and interpreting input from sequences controlled by a stream buffer.

Two groups of member function signatures share common properties: the formatted input functions (or extractors) and the unformatted input functions. Both groups of input functions are described as if they obtain (or extract) input characters by calling rdbuf()→sbumpc() or rdbuf()→sgetc(). They may use other public members of istream.

If rdbuf()→sbumpc() or rdbuf()→sgetc() returns traits::eof(), then the input function, except as explicitly noted otherwise, completes its actions and does setstate(eofbit), which may throw ios_base::failure (29.5.5.4), before returning.

If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets badbit in the error state. If badbit is set in exceptions(), the input function rethrows the exception without completing its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.

### 29.7.4.2.2 Constructors

```cpp
explicit basic_istream(basic_streambuf<charT, traits>* sb);
```

**Effects:** Initializes the base class subobject with basic_ios::init(sb) (29.5.5.2).

**Postconditions:** gcount() == 0.

```cpp
basic_istream(basic_istream&& rhs);
```

**Effects:** Default constructs the base class, copies the gcount() from rhs, calls basic_ios<charT, traits>::move(rhs) to initialize the base class, and sets the gcount() for rhs to 0.

**Remarks:** Does not perform any operations of rdbuf().

### 29.7.4.2.3 Assignment and swap

```cpp
basic_istream& operator=(basic_istream&& rhs);
```

**Effects:** Equivalent to: swap(rhs).

**Returns:** *this.

```cpp
void swap(basic_istream& rhs);
```

**Effects:** Calls basic_ios<charT, traits>::swap(rhs). Exchanges the values returned by gcount() and rhs.gcount().

### 29.7.4.2.4 Class basic_istream::sentry

```cpp
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_istream<charT, traits>::sentry {
    bool ok_; // exposition only
  public:
    explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false);
    ~sentry();
    explicit operator bool() const { return ok_; }
    sentry(const sentry&) = delete;
    sentry& operator=(const sentry&) = delete;
  };
}
```

The class sentry defines a class that is responsible for doing exception safe prefix and suffix operations.

```cpp
explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false);
```

**Effects:** If is.good() is false, calls is.setstate(failbit). Otherwise, prepares for formatted or unformatted input. First, if is.tie() is not a null pointer, the function calls is.tie()→flush() to synchronize the output sequence with any associated external C stream. Except that this call can be suppressed if the put area of is.tie() is empty. Further an implementation is allowed to defer the call to flush until a call of is.rdbuf()→underflow() occurs. If no such call occurs before
the `sentry` object is destroyed, the call to `flush` may be eliminated entirely.\(^{304}\) If `noskipws` is zero and `is.flags() & ios_base::skipws` is nonzero, the function extracts and discards each character as long as the next available input character `c` is a whitespace character. If `is.rdbuf()->sbumpc()` or `is.rdbuf()->sgetc()` returns `traits::eof()`, the function calls `setstate(failbit | eofbit)` (which may throw `ios_base::failure`).

Remarks: The constructor

```cpp
explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false)
```

uses the currently imbued locale in `is`, to determine whether the next input character is whitespace or not.

To decide if the character `c` is a whitespace character, the constructor performs as if it executes the following code fragment:

```cpp
const ctype<charT>& ctype = use_facet<ctype<charT>>(is.getloc());
if (ctype.is(ctype.space, c) != 0)
  // `c` is a whitespace character.
```

If, after any preparation is completed, `is.good()` is `true`, `ok_` ! = `false` otherwise, `ok_` == `false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure`).\(^{305}\)

\[\sim\text{sentry}\];

Effects: None.

```cpp
explicit operator bool() const;
```

Returns: `ok_`.

29.7.4.3 Formatted input functions [istream.formatted]

29.7.4.3.1 Common requirements [istream.formatted(reqmts)]

Each formatted input function begins execution by constructing an object of class `sentry` with the `noskipws` (second) argument `false`. If the `sentry` object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. If an exception is thrown during input then `ios_base::badbit` is turned on\(^{306}\) in `*this`'s error state. If `(exceptions() & badbit) != 0` then the exception is rethrown. In any case, the formatted input function destroys the `sentry` object. If no exception has been thrown, it returns `*this`.

29.7.4.3.2 Arithmetic extractors [istream.formatted.arithmetic]

```cpp
operator>>(unsigned short& val);
operator>>(unsigned int& val);
operator>>(long& val);
operator>>(unsigned long& val);
operator>>(long long& val);
operator>>(unsigned long long& val);
operator>>(float& val);
operator>>(double& val);
operator>>(long double& val);
operator>>(bool& val);
operator>>(void*& val);
```

As in the case of the inserters, these extractors depend on the locale's `num_get<>` (28.4.3.2) object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in 29.7.4.3.1). After a `sentry` object is constructed, the conversion occurs as if performed by the following code fragment:

```cpp
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iosstate err = iostate::goodbit;
use_facet<numget>(loc).get(*this, 0, *this, err, val);
setstate(err);
```

\(^{304}\) This will be possible only in functions that are part of the library. The semantics of the constructor used in user code is as specified.

\(^{305}\) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

\(^{306}\) This is done without causing an `ios_base::failure` to be thrown.
In the above fragment, loc stands for the private member of the basic_ios class.

[Note 1: The first argument provides an object of the istreambuf_iterator class which is an iterator pointed to an input stream. It bypasses istreams and uses streambufs directly. — end note]

Class locale relies on this type as its interface to istream, so that it does not need to depend directly on istream.

```
operator>>(short& val);
```

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iosstate err = ios_base::goodbit;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if (lval < numeric_limits<short>::min()) {
    err |= ios_base::failbit;
    val = numeric_limits<short>::min();
} else if (numeric_limits<short>::max() < lval) {
    err |= ios_base::failbit;
    val = numeric_limits<short>::max();
} else
    val = static_cast<short>(lval);
setstate(err);
```

```
operator>>(int& val);
```

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iosstate err = ios_base::goodbit;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if (lval < numeric_limits<int>::min()) {
    err |= ios_base::failbit;
    val = numeric_limits<int>::min();
} else if (numeric_limits<int>::max() < lval) {
    err |= ios_base::failbit;
    val = numeric_limits<int>::max();
} else
    val = static_cast<int>(lval);
setstate(err);
```

### 29.7.4.3.3 basic_istream::operator>>

```
operator>>(basic_istream<charT, traits>& (*pf)(basic_istream<charT, traits>&));
```

1. **Effects**: None. This extractor does not behave as a formatted input function (as described in 29.7.4.3.1).
2. **Returns**: pf(*this).

```
operator>>(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
```

3. **Effects**: Calls pf(*this). This extractor does not behave as a formatted input function (as described in 29.7.4.3.1).
4. **Returns**: *this.

```
operator>>(ios_base& (*pf)(ios_base&));
```

5. **Effects**: Calls pf(*this). This extractor does not behave as a formatted input function (as described in 29.7.4.3.1).
6. **Returns**: *this.

---

307) See, for example, the function signature ws(basic_istream&) (29.7.4.5).
308) See, for example, the function signature dec(ios_base&) (29.5.6.3).
template<class charT, class traits, size_t N>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& in, charT (&s)[N]);

Effects: Behaves like a formatted input member (as described in 29.7.4.3.1) of in. After a sentry object is constructed, operator>> extracts characters and stores them into s. If width() is greater than zero, n is \(\min(\text{size}_t(\text{width}()), N)\). Otherwise n is N. n is the maximum number of characters stored.

Characters are extracted and stored until any of the following occurs:

1. n-1 characters are stored;
2. end of file occurs on the input sequence;
3. letting \(ct\) be use_facet<ctype<charT>>(in.getloc()), \(ct.is(ct.space, c)\) is true.

operator>> then stores a null byte (\(\text{charT}()\)) in the next position, which may be the first position if no characters were extracted. operator>> then calls width(0).

If the function extracted no characters, it calls setstate(failbit), which may throw ios_base::failure (29.5.5.4).

Returns: in.

template<class charT, class traits>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& in, charT& c);

Effects: Behaves like a formatted input member (as described in 29.7.4.3.1) of in. After a sentry object is constructed a character is extracted from in, if one is available, and stored in c. Otherwise, the function calls in.setstate(failbit).

Returns: in.

basic_istream<charT, traits>& operator>>(basic_streambuf<charT, traits>** sb);

Effects: Behaves as an unformatted input function (29.7.4.4). If \(\text{sb}\) is null, calls setstate(failbit), which may throw ios_base::failure (29.5.5.4). After a sentry object is constructed, extracts characters from *this and inserts them in the output sequence controlled by \(\text{sb}\). Characters are extracted and inserted until any of the following occurs:

1. end-of-file occurs on the input sequence;
2. inserting in the output sequence fails (in which case the character to be inserted is not extracted);
3. an exception occurs (in which case the exception is caught).

If the function inserts no characters, it calls setstate(failbit), which may throw ios_base::failure (29.5.5.4). If it inserted no characters because it caught an exception thrown while extracting characters from *this and failbit is set in exceptions() (29.5.5.4), then the caught exception is rethrown.

Returns: *this.

29.7.4.4 Unformatted input functions

Each unformatted input function begins execution by constructing an object of class sentry with the default argument noskipws (second) argument true. If the sentry object returns true, when converted to a value of type bool, the function endeavors to obtain the requested input. Otherwise, if the sentry constructor exits by throwing an exception or if the sentry object returns false, when converted to a value of type bool, the function returns without attempting to obtain any input. In either case the number of extracted characters is set to 0; unformatted input functions taking a character array of nonzero size as an argument shall also store a null character (using charT()) in the first location of the array. If an exception is thrown during input then
ios_base::badbit is turned on \(^{[309]}\) in \(\ast this\)'s error state. (Exceptions thrown from basic_ios<>::clear() are not caught or rethrown.) If \((\text{exceptions()}\&\text{badbit}) != 0\) then the exception is rethrown. It also counts the number of characters extracted. If no exception has been thrown it ends by storing the count in a member object and returning the value specified. In any event the sentry object is destroyed before leaving the unformatted input function.

```
streamsize gcount() const;
```

2. **Effects**: None. This member function does not behave as an unformatted input function (as described above).

3. **Returns**: The number of characters extracted by the last unformatted input member function called for the object.

```
int_type get();
```

4. **Effects**: Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts a character \(c\), if one is available. Otherwise, the function calls `setstate(failbit)`, which may throw `ios_base::failure (29.5.5.4)`.

5. **Returns**: \(c\) if available, otherwise `traits::eof()`.

```
basic_istream<charT, traits>& get(char_type& c);
```

6. **Effects**: Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts a character \(c\), if one is available, and assigns it to \(c\).\(^{[310]}\) Otherwise, the function calls `setstate(failbit)` (which may throw `ios_base::failure (29.5.5.4)`).

7. **Returns**: `\ast this`.

```
basic_istream<charT, traits>& get(char_type* s, streamsize n, char_type delim);
```

8. **Effects**: Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by \(s\).\(^{[311]}\) Characters are extracted and stored until any of the following occurs:

- \(n\) is less than one or \(n - 1\) characters are stored;
- end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit));
- `traits::eq(c, delim)` for the next available input character \(c\) (in which case \(c\) is not extracted).

9. If the function stores no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure (29.5.5.4)`). In any case, if \(n\) is greater than zero it then stores a null character into the next successive location of the array.

10. **Returns**: `\ast this`.

```
basic_istream<charT, traits>& get(char_type* s, streamsize n);
```

11. **Effects**: Calls `get(s, n, widen(\'n\'))`.

12. **Returns**: Value returned by the call.

```
basic_istream<charT, traits>& get(basic_streambuf<char_type, traits>& sb, char_type delim);
```

13. **Effects**: Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and inserts them in the output sequence controlled by \(sb\). Characters are extracted and inserted until any of the following occurs:

- end-of-file occurs on the input sequence;
- inserting in the output sequence fails (in which case the character to be inserted is not extracted);
- `traits::eq(c, delim)` for the next available input character \(c\) (in which case \(c\) is not extracted);
- an exception occurs (in which case, the exception is caught but not rethrown).

14. If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure (29.5.5.4)`.

\(^{[309]}\) This is done without causing an `ios_base::failure` to be thrown.

\(^{[310]}\) Note that this function is not overloaded on types `signed char` and `unsigned char`.

\(^{[311]}\) Note that this function is not overloaded on types `signed char` and `unsigned char`.

\(\text{§ 29.7.4.4}\)
Returns: \*this.

basic_istream<charT, traits>& get(basic_streambuf<char_type, traits>& sb);

Effects: Calls get(sb, widen(‘\n’)).

Returns: Value returned by the call.

basic_istream<charT, traits>& getline(char_type* s, streamsize n, char_type delim);

Effects: Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by \*s. \(^{312}\) Characters are extracted and stored until one of the following occurs:
1. end-of-file occurs on the input sequence (in which case the function calls \texttt{setstate(eofbit)});
2. \texttt{traits::eq(c, delim)} for the next available input character \(c\) (in which case the input character is extracted but not stored); \(^{313}\)
3. \(n\) is less than one or \(n - 1\) characters are stored (in which case the function calls \texttt{setstate(failbit)}).

These conditions are tested in the order shown. \(^{314}\)

If the function extracts no characters, it calls \texttt{setstate(failbit)} (which may throw \texttt{ios_base::failure \((29.5.5.4)\)}). \(^{315}\)

In any case, if \(n\) is greater than zero, it then stores a null character (using \texttt{charT()} into the next successive location of the array.

Returns: \*this.

[Example 1:]

```cpp
#include <iostream>

int main() {
  using namespace std;
  const int line_buffer_size = 100;

  char buffer[line_buffer_size];
  int line_number = 0;
  while (cin.getline(buffer, line_buffer_size, ‘\n’) || cin.gcount()) {
    int count = cin.gcount();
    if (cin.eof())
      cout << "Partial final line"; // cin.fail() is false
    else if (cin.fail()) {
      cout << "Partial long line";
      cin.clear(cin.rdstate() & ~ios_base::failbit);
    } else {
      count--; // Don’t include newline in count
      cout << "Line " << ++line_number;
    }
    cout << " (" << count << " chars): " << buffer << endl;
  }
}
```

basic_istream<charT, traits>& getline(char_type* s, streamsize n);

Returns: getline(s, n, widen(‘\n’))

\(^{312}\) Note that this function is not overloaded on types \texttt{signed char} and \texttt{unsigned char}.

\(^{313}\) Since the final input character is “extracted”, it is counted in the \texttt{gcount()}, even though it is not stored.

\(^{314}\) This allows an input line which exactly fills the buffer, without setting \texttt{failbit}. This is different behavior than the historical AT&T implementation.

\(^{315}\) This implies an empty input line will not cause \texttt{failbit} to be set.
basic_istream<charT, traits>& ignore(streamsize n = 1, int_type delim = traits::eof());

**Effects:** Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and discards them. Characters are extracted until any of the following occurs:

- \( n \neq \text{numeric\_limits<streamsize>::max()} \) (17.3.5) and \( n \) characters have been extracted so far

- end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`, which may throw `ios_base::failure` (29.5.5.4));

- `traits::eq_int_type(trait::to_int_type(c), delim)` for the next available input character `c` (in which case `c` is extracted).

[Note 1: The last condition will never occur if `traits::eq_int_type(delim, traits::eof())`. — end note]

**Returns:** *this.

int_type peek();

**Effects:** Behaves as an unformatted input function (as described above). After constructing a sentry object, reads but does not extract the current input character.

**Returns:** `traits::eof()` if `good()` is `false`. Otherwise, returns `rdbuf()->sgetc()`.

basic_istream<charT, traits>& read(char_type* s, streamsize n);

**Effects:** Behaves as an unformatted input function (as described above). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. Otherwise extracts characters and stores them into successive locations of an array whose first element is designated by `s`. Characters are extracted and stored until either of the following occurs:

- \( n \) characters are stored;
- end-of-file occurs on the input sequence (in which case the function calls `setstate(failbit | eofbit)`, which may throw `ios_base::failure` (29.5.5.4)).

**Returns:** *this.

streamsize readsome(char_type* s, streamsize n);

**Effects:** Behaves as an unformatted input function (as described above). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. Otherwise extracts characters and stores them into successive locations of an array whose first element is designated by `s`. If `rdbuf()->in_avail() == -1`, calls `setstate(eofbit)` (which may throw `ios_base::failure` (29.5.5.4)), and extracts no characters;

- If `rdbuf()->in_avail() == 0`, extracts no characters
- If `rdbuf()->in_avail() > 0`, extracts `\min(rdbuf()->in_avail(), n)`.

**Returns:** The number of characters extracted.

basic_istream<charT, traits>& putback(char_type c);

**Effects:** Behaves as an unformatted input function (as described above), except that the function first clears `eofbit`. After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. If `rdbuf()` is not null, calls `rdbuf()->sputbackc(c)`. If `rdbuf()` is null, or if `sputbackc` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (29.5.5.4)).

[Note 2: This function extracts no characters, so the value returned by the next call to `gcount()` is 0. — end note]

**Returns:** *this.

basic_istream<charT, traits>& unget();

**Effects:** Behaves as an unformatted input function (as described above), except that the function first clears `eofbit`. After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. If `rdbuf()` is not null, calls `rdbuf()->sungetc()`. If `rdbuf()` is null, or if `sungetc` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (29.5.5.4)).

316) Note that this function is not overloaded on types `signed char` and `unsigned char`.  

§ 29.7.4.4 1412
is null, or if `sungetc` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `std::ios_base::failure` (29.5.5.4)).

[Note 3: This function extracts no characters, so the value returned by the next call to `gcount()` is 0. — end note]

Returns: *this.

36
int sync();

Effects: Behaves as an unformatted input function (as described above), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `rdbuf()` is a null pointer, returns -1. Otherwise, calls `rdbuf()->pubsync()` and, if that function returns -1 calls `setstate(badbit)` (which may throw `std::ios_base::failure` (29.5.5.4)), and returns -1. Otherwise, returns zero.

37

pos_type tellg();

Effects: Behaves as an unformatted input function (as described above), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`.

Returns: After constructing a sentry object, if `fail()` != false, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, in)`.

38

basic_istream<charT, traits>& seekg(pos_type pos);

Effects: Behaves as an unformatted input function (as described above), except that the function first clears `eofbit`, it does not count the number of characters extracted, and it does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `fail()` != true, executes `rdbuf()->pubseekoff(pos, ios_base::in)`. In case of failure, the function calls `setstate(failbit)` (which may throw `std::ios_base::failure`).

Returns: *this.

39

basic_istream<charT, traits>& seekg(off_type off, ios_base::seekdir dir);

Effects: Behaves as an unformatted input function (as described above), except that the function first clears `eofbit`, does not count the number of characters extracted, and it does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `fail()` != true, executes `rdbuf()->pubseekoff(off, dir, ios_base::in)`. In case of failure, the function calls `setstate(failbit)` (which may throw `std::ios_base::failure`).

Returns: *this.

29.7.4.5 Standard basic_istream manipulators

Each instantiation of the function template specified in this subclause is a designated addressable function (16.4.5.2.1).

template<class charT, class traits>
basic_istream<charT, traits>& ws(basic_istream<charT, traits>& is);

Effects: Behaves as an unformatted input function (29.7.4.4), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `is.gcount()`. After constructing a sentry object extracts characters as long as the next available character is whitespace or until there are no more characters in the sequence. Whitespace characters are distinguished with the same criterion as used by `sentry::sentry` (29.7.4.2.4). If `ws` stops extracting characters because there are no more available it sets `eofbit`, but not `failbit`.

Returns: is.

29.7.4.6 Rvalue stream extraction

template<class Istream, class T>
Istream&& operator>>(Istream&& is, T&& x);

Constraints: The expression `is >> std::forward<T>(x)` is well-formed when treated as an unevaluated operand and `Istream` is publicly and unambiguously derived from `std::ios_base`.

Effects: Equivalent to:
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_iostream
        : public basic_istream<charT, traits>,
          public basic_ostream<charT, traits> {
public:
    using char_type = charT;
    using int_type = typename traits::int_type;
    using pos_type = typename traits::pos_type;
    using off_type = typename traits::off_type;
    using traits_type = traits;

    // 29.7.4.7.2, constructor
    explicit basic_iostream(basic_streambuf<charT, traits>* sb);

    // 29.7.4.7.3, destructor
    virtual ~basic_iostream();

    protected:
        // 29.7.4.7.2, constructor
    basic_iostream(const basic_iostream&) = delete;
    basic_iostream(basic_iostream&& rhs);

        // 29.7.4.7.4, assign and swap
    basic_iostream& operator=(const basic_iostream&) = delete;
    basic_iostream& operator=(basic_iostream&& rhs);
    void swap(basic_iostream& rhs);
    
};

The class template basic_iostream inherits a number of functions that allow reading input and writing output to sequences controlled by a stream buffer.

29.7.4.7.2 Constructors

explicit basic_iostream(basic_streambuf<charT, traits>* sb);

Effects: Initializes the base class subobjects with basic_istream<charT, traits>(sb) (29.7.4.2) and basic_ostream<charT, traits>(sb) (29.7.5.2).

Postconditions: rdbuf() == sb and gcount() == 0.

basic_iostream(basic_iostream&& rhs);

Effects: Move constructs from the rvalue rhs by constructing the basic_istream base class with move(rhs).

29.7.4.7.3 Destructor

virtual ~basic_iostream();

Remarks: Does not perform any operations on rdbuf().

29.7.4.7.4 Assignment and swap

basic_iostream& operator=(basic_iostream&& rhs);

Effects: Equivalent to: swap(rhs).

void swap(basic_iostream& rhs);

Effects: Calls basic_istream<charT, traits>::swap(rhs).
29.7.5  Output streams

29.7.5.1  General

The header `<ostream>` defines a type and several function signatures that control output to a stream buffer along with a function template that inserts into stream rvalues.

29.7.5.2  Class template basic_ostream

29.7.5.2.1  General

namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_ostream : virtual public basic_ios<charT, traits> { 
        public:
            // types (inherited from basic_ios (29.5.5))
            using char_type = charT;
            using int_type = typename traits::int_type;
            using pos_type = typename traits::pos_type;
            using off_type = typename traits::off_type;
            using traits_type = traits;
            // 29.7.5.2.2, constructor/destructor
            explicit basic_ostream(basic_streambuf<char_type, traits>* sb);
            virtual ~basic_ostream();
            // 29.7.5.3, formatted output
            basic_ostream<charT, traits>&
                operator<<(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&));
            basic_ostream<charT, traits>&
                operator<<(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
            basic_ostream<charT, traits>&
                operator<<(ios_base& (*pf)(ios_base&));
            basic_ostream<charT, traits>& operator<<(bool n);
            basic_ostream<charT, traits>& operator<<(short n);
            basic_ostream<charT, traits>& operator<<(unsigned short n);
            basic_ostream<charT, traits>& operator<<(int n);
            basic_ostream<charT, traits>& operator<<(unsigned int n);
            basic_ostream<charT, traits>& operator<<(long n);
            basic_ostream<charT, traits>& operator<<(unsigned long n);
            basic_ostream<charT, traits>& operator<<(long long n);
            basic_ostream<charT, traits>& operator<<(unsigned long long n);
            basic_ostream<charT, traits>& operator<<(float f);
            basic_ostream<charT, traits>& operator<<(double f);
            basic_ostream<charT, traits>& operator<<(const void* p);
            basic_ostream<charT, traits>& operator<<(nullptr_t);
            basic_ostream<charT, traits>& operator<<(basic_streambuf<char_type, traits>* sb);
            // 29.7.5.4, unformatted output
            basic_ostream<charT, traits>& put(char_type c);
            basic_ostream<charT, traits>& write(const char_type* s, streamsize n);
            basic_ostream<charT, traits>& flush();
            // 29.7.5.2.5, seeks
            pos_type.tellp();
            basic_ostream<charT, traits>& seekp(pos_type);
            basic_ostream<charT, traits>& seekp(off_type, ios_base::seekdir);
protected:

// 29.7.5.2.2, copy/move constructor
basic_ostream(const basic_ostream&) = delete;
basic_ostream(basic_ostream&& rhs);  

// 29.7.5.2.3, assign and swap
basic_ostream& operator=(const basic_ostream&) = delete;
basic_ostream& operator=(basic_ostream&& rhs);
void swap(basic_ostream& rhs);  

};

// 29.7.5.3.4, character inserters
template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, charT);
template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, char);  
template<class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, char);

template<class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, signed char);  
template<class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, unsigned char);  

template<class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, wchar_t) = delete;
template<class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, char8_t) = delete;

template<class traits>
basic_ostream<wchar_t, traits>& operator<<(basic_ostream<wchar_t, traits>&, char8_t) = delete;

template<class traits>
basic_ostream<wchar_t, traits>& operator<<(basic_ostream<wchar_t, traits>&, char16_t) = delete;

template<class traits>
basic_ostream<wchar_t, traits>& operator<<(basic_ostream<wchar_t, traits>&, char32_t) = delete;

template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, const charT*);  
template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, const char*);  

template<class charT, class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, const signed char*);  

template<class charT, class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, const unsigned char*);  

template<class charT, class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, const wchar_t*);  

template<class charT, class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, const char8_t*);  

template<class charT, class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, const char16_t*);  

§ 29.7.5.2.1
template<class traits>
    basic_ostream<char, traits>&
    operator<<(basic_ostream<char, traits>&, const char32_t*) = delete;

template<class traits>
    basic_ostream<wchar_t, traits>&
    operator<<(basic_ostream<wchar_t, traits>&, const char8_t*) = delete;

template<class traits>
    basic_ostream<wchar_t, traits>&
    operator<<(basic_ostream<wchar_t, traits>&, const char16_t*) = delete;

template<class traits>
    basic_ostream<wchar_t, traits>&
    operator<<(basic_ostream<wchar_t, traits>&, const char32_t*) = delete;

1. The class template basic_ostream defines a number of member function signatures that assist in formatting and writing output to output sequences controlled by a stream buffer.
2. Two groups of member function signatures share common properties: the formatted output functions (or inserters) and the unformatted output functions. Both groups of output functions generate (or insert) output characters by actions equivalent to calling rdbuf()->sputc(int_type). They may use other public members of basic_ostream except that they shall not invoke any virtual members of rdbuf() except overflow(), xsputn(), and sync().
3. If one of these called functions throws an exception, then unless explicitly noted otherwise the output function sets badbit in the error state. If badbit is set in exceptions(), the output function rethrows the exception without completing its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.
4. [Note 1: The deleted overloads of operator<< prevent formatting characters as integers and strings as pointers. — end note]

### 29.7.5.2.2 Constructors

```cpp
explicit basic_ostream(basic_streambuf<charT, traits>* sb);
```
1. **Effects**: Initializes the base class subobject with basic_ios<charT, traits>::init(sb) [29.5.5.2].
2. **Postconditions**: rdbuf() == sb.

```cpp
basic_ostream(basic_ostream&& rhs);
```
3. **Effects**: Move constructs from the rvalue rhs. This is accomplished by default constructing the base class and calling basic_ios<charT, traits>::move(rhs) to initialize the base class.

```cpp
virtual ~basic_ostream();
```
4. **Remarks**: Does not perform any operations on rdbuf().

### 29.7.5.2.3 Assignment and swap

```cpp
basic_ostream& operator=(basic_ostream&& rhs);
```
1. **Effects**: Equivalent to: swap(rhs).
2. **Returns**: *this.

```cpp
void swap(basic_ostream& rhs);
```
3. **Effects**: Calls basic_ios<charT, traits>::swap(rhs).

### 29.7.5.2.4 Class basic_ostream::sentry

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_ostream<charT, traits>::sentry {
        bool ok_; // exposition only

        public:
            explicit sentry(basic_ostream<charT, traits>& os);
            ~sentry();
            explicit operator bool() const { return ok_; }
    }
}
```
The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

```cpp
explicit sentry(const sentry&) = delete;
sentry& operator=(const sentry&) = delete;
```

1. `explicit sentry(basic_ostream<charT, traits>& os);`
   
   *Effects:* Returns `ok_`.

2. If `os.good()` is nonzero, prepares for formatted or unformatted output. If `os.tie()` is not a null pointer, calls `os.tie()->flush()`.

3. If, after any preparation is completed, `os.good()` is `true`, `ok_ == true` otherwise, `ok_ == false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure`).

```cpp
~sentry();
```

4. If `(os.flags() & ios_base::unitbuf) && !uncaught_exceptions() && os.good()` is `true`, calls `os.rdbuf()->pubsync()`. If that function returns `-1`, sets `badbit` in `os.rdstate()` without propagating an exception.

```cpp
explicit operator bool() const;
```


### 29.7.5.2.5 Seek members

Each seek member function begins execution by constructing an object of class `sentry`. It returns by destroying the `sentry` object.

```cpp
pos_type tellp();
```

6. *Returns:* If `fail() != false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, out)`.

```cpp
basic_ostream<charT, traits>& seekp(pos_type pos);
```

6. *Effects:* If `fail() != true`, executes `rdbuf()->pubseekpos(pos, ios_base::out)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

```cpp
basic_ostream<charT, traits>& seekp(off_type off, ios_base::seekdir dir);
```

6. *Effects:* If `fail() != true`, executes `rdbuf()->pubseekoff(off, dir, ios_base::out)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

### 29.7.5.3 Formatted output functions

1. Each formatted output function begins execution by constructing an object of class `sentry`. If this object returns `true` when converted to a value of type `bool`, the function endeavors to generate the requested output. If the generation fails, then the formatted output function does `setstate(ios_base::failbit)`, which can throw an exception. If an exception is thrown during output, then `ios_base::badbit` is turned on in this object’s error state. If `exceptions() & badbit` != 0 then the exception is rethrown. Whether or not an exception is thrown, the `sentry` object is destroyed before leaving the formatted output function. If no exception is thrown, the result of the formatted output function is `*this`.

2. The descriptions of the individual formatted output functions describe how they perform output and do not mention the `sentry` object.

3. If a formatted output function of a stream `os` determines padding, it does so as follows. Given a `charT` character sequence `seq` where `charT` is the character type of the stream, if the length of `seq` is less than `os.width()`, then enough copies of `os.fill()` are added to this sequence as necessary to pad to a width

---

317) The call `os.tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.

318) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

319) This is done without causing an `ios_base::failure` to be thrown.
os.width() characters. If \((\text{os.flags()} \& \text{ios::adjustfield}) == \text{ios::left}\) is true, the fill characters are placed after the character sequence; otherwise, they are placed before the character sequence.

### 29.7.5.3.2 Arithmetic inserters

[ostream.inserters.arithmetic]

```cpp
operator<<(bool val);
operator<<(short val);
operator<<(unsigned short val);
operator<<(int val);
operator<<(unsigned int val);
operator<<(long val);
operator<<(unsigned long val);
operator<<(long long val);
operator<<(unsigned long long val);
operator<<(float val);
operator<<(double val);
operator<<(long double val);
operator<<(const void* val);
```

**Effects:** The classes `num_get<>` and `num_put<>` handle locale-dependent numeric formatting and parsing. These inserter functions use the imbued `locale` value to perform numeric formatting. When val is of type `bool`, `long`, `unsigned long`, `long long`, `unsigned long long`, `double`, `long double`, or `const void*`, the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
  >(getloc()).put(*this, *this, fill(), val).failed();
```

When val is of type `short` the formatting conversion occurs as if it performed the following code fragment:

```cpp
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
  >(getloc()).put(*this, *this, fill(),
  baseflags == ios_base::oct || baseflags == ios_base::hex
  ? static_cast<long>(static_cast<unsigned short>(val))
  : static_cast<long>(val)).failed();
```

When val is of type `int` the formatting conversion occurs as if it performed the following code fragment:

```cpp
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
  >(getloc()).put(*this, *this, fill(),
  baseflags == ios_base::oct || baseflags == ios_base::hex
  ? static_cast<long>(static_cast<unsigned int>(val))
  : static_cast<long>(val)).failed();
```

When val is of type `unsigned short` or `unsigned int` the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
  >(getloc()).put(*this, *this, fill(),
  static_cast<unsigned long>(val)).failed();
```

When val is of type `float` the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
  >(getloc()).put(*this, *this, fill(),
  static_cast<double>(val)).failed();
```

The first argument provides an object of the `ostreambuf_iterator<>` class which is an iterator for class `basic_ostream<>>. It bypasses `ostreams` and uses `streambufs` directly. Class `locale` relies on these types as its interface to iostreams, since for flexibility it has been abstracted away from direct dependence on `ostream`. The second parameter is a reference to the base class subobject of type...
ios_base. It provides formatting specifications such as field width, and a locale from which to obtain other facets. If failed is true then does setstate(badbit), which may throw an exception, and returns.

3

Returns: *this.

29.7.5.3.3 basic_ostream::operator<<

basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&&));

1

Effects: None. Does not behave as a formatted output function (as described in 29.7.5.3.1).

2

Returns: pf(*this).320

basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& (*pf)(basic_istream<charT, traits>&&));

3

Effects: Calls pf(*this). This inserter does not behave as a formatted output function (as described in 29.7.5.3.1).

4

Returns: *this.321

basic_ostream<charT, traits>& operator<<(*(ios_base& (*pf)(ios_base&));

5

Effects: Calls pf(*this). This inserter does not behave as a formatted output function (as described in 29.7.5.3.1).

6

Returns: *this.

basic_ostream<charT, traits>& operator<<(basic_streambuf<charT, traits>* sb);

7

Effects: Behaves as an unformatted output function (29.7.5.4). After the sentry object is constructed, if sb is null calls setstate(badbit) (which may throw ios_base::failure).

8

Gets characters from sb and inserts them in *this. Characters are read from sb and inserted until any of the following occurs:

(8.1) — end-of-file occurs on the input sequence;

(8.2) — inserting in the output sequence fails (in which case the character to be inserted is not extracted);

(8.3) — an exception occurs while getting a character from sb.

9

If the function inserts no characters, it calls setstate(failbit) (which may throw ios_base::failure (29.5.5.4)). If an exception was thrown while extracting a character, the function sets failbit in the error state, and if failbit is set in exceptions() the caught exception is rethrown.

10

Returns: *this.

basic_ostream<charT, traits>& operator<<(nullptr_t);

11

Effects: Equivalent to:

return *this << s;

where s is an implementation-defined NTCTS (3.36).

29.7.5.3.4 Character inserter function templates

template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, charT c);

template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, char c);

// specialization

template<class traits>
 basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, char c);

// signed and unsigned

template<class traits>
 basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, signed char c);

320) See, for example, the function signature endl(basic_ostream&) (29.7.5.5).
321) See, for example, the function signature dec(ios_base&) (29.5.6.3).
template<class traits>
   basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, unsigned char c);

   Effects: Behaves as a formatted output function (29.7.5.3.1) of out. Constructs a character sequence seq. If c has type char and the character type of the stream is not char, then seq consists of out.widen(c); otherwise seq consists of c. Determines padding for seq as described in 29.7.5.3.1. Inserts seq into out. Calls os.width(0).

   Returns: out.

template<class charT, class traits>
   basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, const charT* s);
   template<class charT, class traits>
   basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, const char* s);
   template<class traits>
   basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, const char* s);
   template<class traits>
   basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, const signed char* s);
   template<class traits>
   basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, const unsigned char* s);

   Preconditions: s is not a null pointer.

   Effects: Behaves like a formatted inserter (as described in 29.7.5.3.1) of out. Creates a character sequence seq of n characters starting at s, each widened using out.widen() (29.5.5.3), where n is the number that would be computed as if by:

   (4.1) traits::length(s) for the overload where the first argument is of type basic_ostream<charT, traits>& and the second is of type const charT*, and also for the overload where the first argument is of type basic_ostream<char, traits>& and the second is of type const char*.

   (4.2) char_traits<char>::length(s) for the overload where the first argument is of type basic_ostream<charT, traits>& and the second is of type const char*.

   (4.3) traits::length(reinterpret_cast<const char*>(s)) for the other two overloads.

   Determines padding for seq as described in 29.7.5.3.1. Inserts seq into out. Calls width(0).

   Returns: out.

29.7.5.4 Unformatted output functions [ostream.unformatted]

Each unformatted output function begins execution by constructing an object of class sentry. If this object returns true, while converting to a value of type bool, the function endeavors to generate the requested output. If an exception is thrown during output, then ios_base::badbit is turned on in *this’s error state. If (exceptions() & badbit) != 0 then the exception is rethrown. In any case, the unformatted output function ends by destroying the sentry object, then, if no exception was thrown, returning the value specified for the unformatted output function.

basic_ostream<charT, traits>& put(char_type c);

   Effects: Behaves as an unformatted output function (as described above). After constructing a sentry object, inserts the character c, if possible.

   Otherwise, calls setstate(badbit) (which may throw ios_base::failure (29.5.5.4)).

   Returns: *this.

basic_ostream& write(const char_type* s, streamsize n);

   Effects: Behaves as an unformatted output function (as described above). After constructing a sentry object, obtains characters to insert from successive locations of an array whose first element is designated by s. Characters are inserted until either of the following occurs:

   (5.1) n characters are inserted;

---

322) This is done without causing an ios_base::failure to be thrown.
323) Note that this function is not overloaded on types signed char and unsigned char.
324) Note that this function is not overloaded on types signed char and unsigned char.
— inserting in the output sequence fails (in which case the function calls `setstate(badbit)`, which may throw `ios_base::failure (29.5.5.4)`).

Returns: *this.

```cpp
basic_ostream& flush();
```

Effects: Behaves as an unformatted output function (as described above). If `rdbuf()` is not a null pointer, constructs a sentry object. If this object returns `true` when converted to a value of type `bool` the function calls `rdbuf()->pubsync()`. If that function returns `-1` calls `setstate(badbit)` (which may throw `ios_base::failure (29.5.5.4)`). Otherwise, if the sentry object returns `false`, does nothing.

Returns: *this.

### 29.7.5.5 Standard manipulators

Each instantiation of any of the function templates specified in this subclause is a designated addressable function (16.4.5.2.1).

#### Template instantiations

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os);
```

Effects: Calls `os.put(os.widen('n'))`, then `os.flush()`.

Returns: `os`.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& ends(basic_ostream<charT, traits>& os);
```

Effects: Inserts a null character into the output sequence: calls `os.put(charT())`.

Returns: `os`.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& flush(basic_ostream<charT, traits>& os);
```

Effects: Calls `os.flush()`.

Returns: `os`.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& emit_on_flush(basic_ostream<charT, traits>& os);
```

Effects: If `os.rdbuf()` is a `basic_syncbuf<charT, traits, Allocator>*`, called `buf` for the purpose of exposition, calls `buf->set_emit_on_sync(true)` (which may throw `ios_base::failure`). Otherwise this manipulator has no effect.

[[Note 1: To work around the issue that the `Allocator` template argument cannot be deduced, implementations can introduce an intermediate base class to `basic_syncbuf` that manages its `emit_on_sync` flag. — end note]]

Returns: `os`.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& noemit_on_flush(basic_ostream<charT, traits>& os);
```

Effects: If `os.rdbuf()` is a `basic_syncbuf<charT, traits, Allocator>*`, called `buf` for the purpose of exposition, calls `buf->set_emit_on_sync(false)`. Otherwise this manipulator has no effect.

Returns: `os`.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& flush_emit(basic_ostream<charT, traits>& os);
```

Effects: Calls `os.flush()`. Then, if `os.rdbuf()` is a `basic_syncbuf<charT, traits, Allocator>*`, called `buf` for the purpose of exposition, calls `buf->emit()`.

Returns: `os`.

### 29.7.5.6 Rvalue stream insertion

```cpp
template<class Ostream, class T>
Ostream&& operator<<(Ostream&& os, const T& x);
```

Constraints: The expression `os << x` is well-formed when treated as an unevaluated operand and `Ostream` is publicly and unambiguously derived from `ios_base`.

§ 29.7.5.6 1422
Effects: As if by: `os << x;`

Returns: `std::move(os);`

29.7.6 Standard manipulators

The header `<iomanip>` defines several functions that support extractors and inserters that alter information maintained by class `ios_base` and its derived classes.

Unspecified `resetiosflags(ios_base::fmtflags mask);`

Returns: An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << resetiosflags(mask)` behaves as if it called `f(out, mask)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> resetiosflags(mask)` behaves as if it called `f(in, mask)`, where the function `f` is defined as:

```cpp
void f(ios_base& str, ios_base::fmtflags mask) {
  // reset specified flags
  str.setf(ios_base::fmtflags(0), mask);
}
```

The expression `out << resetiosflags(mask)` has type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> resetiosflags(mask)` has type `basic_istream<charT, traits>&` and value in.

Unspecified `setiosflags(ios_base::fmtflags mask);`

Returns: An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << setiosflags(mask)` behaves as if it called `f(out, mask)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> setiosflags(mask)` behaves as if it called `f(in, mask)`, where the function `f` is defined as:

```cpp
void f(ios_base& str, ios_base::fmtflags mask) {
  // set specified flags
  str.setf(mask);
}
```

The expression `out << setiosflags(mask)` has type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> setiosflags(mask)` has type `basic_istream<charT, traits>&` and value in.

Unspecified `setbase(int base);`

Returns: An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << setbase(base)` behaves as if it called `f(out, base)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> setbase(base)` behaves as if it called `f(in, base)`, where the function `f` is defined as:

```cpp
void f(ios_base& str, int base) {
  // set basefield
  str.setf(base == 8 ? ios_base::oct :
    base == 10 ? ios_base::dec :
    base == 16 ? ios_base::hex :
    ios_base::fmtflags(0), ios_base::basefield);
}
```

The expression `out << setbase(base)` has type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> setbase(base)` has type `basic_istream<charT, traits>&` and value in.

Unspecified `setfill(char_type c);`

Returns: An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` and `c` has type `charT` then the expression `out << setfill(c)` behaves as if it called `f(out, c)`, where the function `f` is defined as:

325 The expression `cin >> resetiosflags(ios_base::skipws)` clears `ios_base::skipws` in the format flags stored in the `basic_istream<charT, traits>` object `cin` (the same as `cin >> noskipws`), and the expression `cout << resetiosflags(ios_base::showbase)` clears `ios_base::showbase` in the format flags stored in the `basic_ostream<charT, traits>` object `cout` (the same as `cout << noshowbase`).

§ 29.7.6 1423
template<class charT, class traits>
void f(basic_ios<charT, traits>& str, charT c) {
    // set fill character
    str.fill(c);
}

The expression out << setfill(c) has type basic_ostream<charT, traits>& and value out.

unsieved setprecision(int n);

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> then the expression out << setprecision(n) behaves as if it called f(out, n), or if in is an object of type basic_istream<charT, traits> then the expression in >> setprecision(n) behaves as if it called f(in, n), where the function f is defined as:

void f(ios_base& str, int n) {
    // set precision
    str.precision(n);
}

The expression out << setprecision(n) has type basic_ostream<charT, traits>& and value out. The expression in >> setprecision(n) has type basic_istream<charT, traits>& and value in.

unsieved setw(int n);

Returns: An object of unspecified type such that if out is an instance of basic_ostream<charT, traits> then the expression out << setw(n) behaves as if it called f(out, n), or if in is an object of type basic_istream<charT, traits> then the expression in >> setw(n) behaves as if it called f(in, n), where the function f is defined as:

void f(ios_base& str, int n) {
    // set width
    str.width(n);
}

The expression out << setw(n) has type basic_ostream<charT, traits>& and value out. The expression in >> setw(n) has type basic_istream<charT, traits>& and value in.

29.7.7 Extended manipulators [ext.manip]

The header <iomanip> defines several functions that support extractors and inserters that allow for the parsing and formatting of sequences and values for money and time.

template<class moneyT> unspecified get_money(moneyT& mon, bool Intl = false);

Mandates: The type moneyT is either long double or a specialization of the basic_string template (Clause 21).

Effects: The expression in >> get_money(mon, Intl) described below behaves as a formatted input function (29.7.4.3.1).

Returns: An object of unspecified type such that if in is an object of type basic_istream<charT, traits> then the expression in >> get_money(mon, Intl) behaves as if it called f(in, mon, Intl), where the function f is defined as:

template<class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, moneyT& mon, bool Intl) {
    using Iter = istreambuf_iterator<charT, traits>
    using MoneyGet = money_get<charT, Iter>

    ios_base::iostate err = ios_base::goodbit;
    const MoneyGet& mg = use_facet<MoneyGet>(str.getloc());

    mg.get(Iter(str.rdbuf()), Iter(), Intl, str, err, mon);

    if (ios_base::goodbit != err)
        str.setstate(err);
}
The expression in \texttt{\textgreater\textgreater get\_money(mon, intl)} has type \texttt{basic\_istream<charT, traits>\&} and value in.

\footnotesize
\texttt{template<class moneyT> unspecified put\_money(const moneyT& mon, bool intl = false);}\

\textbf{Mandates:} The type \texttt{moneyT} is either \texttt{long double} or a specialization of the \texttt{basic\_string} template (Clause 21).

\textbf{Returns:} An object of unspecified type such that if \texttt{out} is an object of type \texttt{basic\_ostream<charT, traits>} then the expression \texttt{\textless\textless put\_money(mon, intl)} behaves as a formatted output function (29.7.5.3.1) that calls \texttt{f(out, mon, intl)}, where the function \texttt{f} is defined as:

\footnotesize
\texttt{template<class charT, class traits, class moneyT>}
\texttt{void f(basic\_ios<charT, traits>& str, const moneyT& mon, bool intl) \{}
\texttt{    using Iter = ostreambuf\_iterator<charT, traits>;}\n\texttt{    using MoneyPut = money\_put<charT, Iter>;}\n\texttt{    const MoneyPut& mp = use\_facet\_money\_put\_charT, Iter\};\n\texttt{    const Iter end = mp.put(Iter(str.rdbuf()), intl, str, str.fill(), mon);}\n\texttt{    if (end.failed())}
\texttt{        str.setstate(ios\_base::badbit);}\n\texttt{\} }

The expression \texttt{\textless\textless put\_money(mon, intl)} has type \texttt{basic\_ostream<charT, traits>\&} and value out.

\footnotesize
\texttt{template<class charT> unspecified get\_time(struct tm* tmb, const charT* fmt);}\

\textbf{Preconditions:} The argument \texttt{tmb} is a valid pointer to an object of type \texttt{struct tm}, and \texttt{[fmt, fmt + char\_traits<charT>::length(fmt)]} is a valid range.

\textbf{Returns:} An object of unspecified type such that if \texttt{in} is an object of type \texttt{basic\_istream<charT, traits>} then the expression in \texttt{\textgreater\textgreater get\_time(tmb, fmt)} behaves as if it called \texttt{f(in, tmb, fmt)}, where the function \texttt{f} is defined as:

\footnotesize
\texttt{template<class charT, class traits>}
\texttt{void f(basic\_ios<charT, traits>& str, struct tm* tmb, const charT* fmt) \{}
\texttt{    using Iter = istreambuf\_iterator<charT, traits>;}\n\texttt{    using TimeGet = time\_get<charT, Iter>;}\n\texttt{    ios\_base::iostate err = ios\_base::goodbit;}\n\texttt{    const TimeGet& tg = use\_facet\_time\_get\(\)(str.getloc());}
\texttt{    tg.get(Iter(str.rdbuf()), Iter(), str, err, tmb, fmt, fmt + traits::length(fmt));}
\texttt{    if (err != ios\_base::goodbit)}
\texttt{        str.setstate(err);}\n\texttt{\} }

The expression \texttt{\textgreater\textgreater get\_time(tmb, fmt)} has type \texttt{basic\_istream<charT, traits>\&} and value in.

\footnotesize
\texttt{template<class charT> unspecified put\_time(const struct tm* tmb, const charT* fmt);}\

\textbf{Preconditions:} The argument \texttt{tmb} is a valid pointer to an object of type \texttt{struct tm}, and \texttt{[fmt, fmt + char\_traits<charT>::length(fmt)]} is a valid range.

\textbf{Returns:} An object of unspecified type such that if \texttt{out} is an object of type \texttt{basic\_ostream<charT, traits>} then the expression \texttt{\textless\textless put\_time(tmb, fmt)} behaves as if it called \texttt{f(out, tmb, fmt)}, where the function \texttt{f} is defined as:

\footnotesize
\texttt{template<class charT, class traits>}
\texttt{void f(basic\_ios<charT, traits>& str, const struct tm* tmb, const charT* fmt) \{}
\texttt{    using Iter = ostreambuf\_iterator<charT, traits>;}\n\texttt{    using TimePut = time\_put<charT, Iter>;}\n\texttt{    const TimePut& tp = use\_facet\_time\_put\(\)(str.getloc());}
The expression `out << put_time(tmb, fmt)` has type `basic_ostream<charT, traits>&` and value `out`.

### 29.7.8 Quoted manipulators

**[quoted.manip]**

1. **[Note 1]**: Quoted manipulators provide string insertion and extraction of quoted strings (for example, XML and CSV formats). Quoted manipulators are useful in ensuring that the content of a string with embedded spaces remains unchanged if inserted and then extracted via stream I/O. — end note

```cpp
template<class charT>
unspecified quoted(const charT* s, charT delim = charT('"'), charT escape = charT('\'));
```

2. **Returns**: An object of unspecified type such that if `out` is an instance of `basic_ostream` with member type `char_type` the same as `charT` and with member type `traits_type`, which in the second and third forms is the same as `traits`, then the expression `out << quoted(s, delim, escape)` behaves as a formatted output function (29.7.5.3.1) of `out`. This forms a character sequence `seq`, initially consisting of the following elements:

   1. **(2.1)** `delim`.
   2. **(2.2)** Each character in `s`. If the character to be output is equal to `escape` or `delim`, as determined by `traits_type::eq`, first output `escape`.
   3. **(2.3)** `delim`.

Let `x` be the number of elements initially in `seq`. Then padding is determined for `seq` as described in 29.7.5.3.1. `seq` is inserted as if by calling `out.rdbuf()->sputn(seq, n)`, where `n` is the larger of `out.width()` and `x`, and `out.width(0)` is called. The expression `out << quoted(s, delim, escape)` has type `basic_ostream<charT, traits>&` and value `out`.

```cpp
template<class charT, class traits, class Allocator>
unspecified quoted(basic_string<charT, traits, Allocator>& s, charT delim = charT('"'), charT escape = charT('\'));
```

3. **Returns**: An object of unspecified type such that:

   1. **(3.1)** If `in` is an instance of `basic_istream` with member types `char_type` and `traits_type` the same as `charT` and `traits`, respectively, then the expression `in >> quoted(s, delim, escape)` behaves as if it extracts the following characters from `in` using `operator>>(basic_istream<charT, traits>&, charT&)` (29.7.4.3.3) which may throw `ios_base::failure` (29.5.3.2.1):

      1. **(3.1.1)** If the first character extracted is equal to `delim`, as determined by `traits_type::eq`, then:
         1. **(3.1.1.1)** Turn off the `skipws` flag.
         1. **(3.1.1.2)** `s.clear()`.
         1. **(3.1.1.3)** Until an unescaped `delim` character is reached or `!in`, extract characters from `in` and append them to `s`, except that if an `escape` is reached, ignore it and append the next character to `s`.
         1. **(3.1.1.4)** Discard the final `delim` character.
         1. **(3.1.1.5)** Restore the `skipws` flag to its original value.
      1. **(3.1.2)** Otherwise, `in >> s`. 

§ 29.7.8
If `out` is an instance of `basic_ostream` with member types `char_type` and `traits_type` the same as `charT` and `traits`, respectively, then the expression `out << quoted(s, delim, escape)` behaves as specified for the `const basic_string<charT, traits, Allocator>&` overload of the quoted function.

The expression `in >> quoted(s, delim, escape)` has type `basic_istream<charT, traits>&` and value in.

The expression `out << quoted(s, delim, escape)` has type `basic_ostream<charT, traits>&` and value out.

### 29.8 String-based streams

**[string.streams]**

#### 29.8.1 Header `<sstream>` synopsis

**[sstream.syn]**

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_stringbuf;
    using stringbuf = basic_stringbuf<char>;
    using wstringbuf = basic_stringbuf<wchar_t>;

    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_istringstream;
    using istringstream = basic_istringstream<char>;
    using wistringstream = basic_istringstream<wchar_t>;

    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_ostringstream;
    using ostringstream = basic_ostringstream<char>;
    using wostringstream = basic_ostringstream<wchar_t>;

    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_stringstream;
    using stringstream = basic_stringstream<char>;
    using wstringstream = basic_stringstream<wchar_t>;
}
```

The header `<sstream>` defines four class templates and eight types that associate stream buffers with objects of class `basic_string`, as described in 21.3.

#### 29.8.2 Class template `basic_stringbuf`

**[stringbuf]**

#### 29.8.2.1 General

**[stringbuf.general]**

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_stringbuf : public basic_streambuf<charT, traits> {
public:
    using char_type = charT;
    using int_type = typename traits::int_type;
    using pos_type = typename traits::pos_type;
    using off_type = typename traits::off_type;
    using traits_type = traits;
    using allocator_type = Allocator;

    // 29.8.2.2, constructors
    basic_stringbuf() : basic_stringbuf(ios_base::in | ios_base::out) {}
    explicit basic_stringbuf(ios_base::openmode which);
}
```
explicit basic_stringbuf(
    const basic_string<CharT, traits, Allocator>& s,
    ios_base::openmode which = ios_base::in | ios_base::out);
explicit basic_stringbuf(const Allocator& a)
    : basic_stringbuf(ios_base::in | ios_base::out, a) {}
basic_stringbuf(ios_base::openmode which, const Allocator& a);
explicit basic_stringbuf(
    basic_string<CharT, traits, Allocator>&& s,
    ios_base::openmode which = ios_base::in | ios_base::out);
template<class SAlloc>
    basic_stringbuf(
        const basic_string<CharT, traits, SAlloc>& s, const Allocator& a)
        : basic_stringbuf(s, ios_base::in | ios_base::out, a) {}
template<class SAlloc>
    basic_stringbuf(
        const basic_string<CharT, traits, SAlloc>& s,
        ios_base::openmode which, const Allocator& a);

    // 29.8.2.3, assign and swap
    basic_stringbuf& operator=(const basic_stringbuf&) = delete;
    basic_stringbuf& operator=(basic_stringbuf&& rhs);
    void swap(basic_stringbuf& rhs) noexcept(
        see below);

    // 29.8.2.4, getters and setters
    allocator_type get_allocator() const noexcept;
    basic_string<CharT, traits, Allocator> str() const &;
    template<class SAlloc>
        basic_string<CharT, traits, SAlloc> str(const SAlloc& sa) const;
    basic_string_view<CharT, traits> view() const noexcept;
    void str(const basic_string<CharT, traits, Allocator>& s);
    template<class SAlloc>
        void str(const basic_string<CharT, traits, SAlloc>& s);
    void str(basic_string<CharT, traits, Allocator>&& s);

protected:
    // 29.8.2.5, overridden virtual functions
    int_type underflow() override;
    int_type pbackfail(int_type c = traits::eof()) override;
    int_type overflow (int_type c = traits::eof()) override;
    basic_streambuf<CharT, traits>* setbuf(charT*, streamsize) override;
    pos_type seekoff(off_type off, ios_base::seekdir way,
                     ios_base::openmode which = ios_base::in | ios_base::out) override;
    pos_type seekpos(pos_type sp,
                     ios_base::openmode which = ios_base::in | ios_base::out) override;

private:
    ios_base::openmode mode; // exposition only
    basic_string<CharT, traits, Allocator> buf; // exposition only
    void init_buf_ptrs(); // exposition only
);
The class `basic_stringbuf` is derived from `basic_streambuf` to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary characters. The sequence can be initialized from, or made available as, an object of class `basic_string`.

For the sake of exposition, the maintained data and internal pointer initialization is presented here as:

1. `ios_base::openmode mode`, has `in` set if the input sequence can be read, and `out` set if the output sequence can be written.
2. `basic_string<CharT, traits, Allocator> buf` contains the underlying character sequence.
3. `init_buf_ptrs()` sets the base class’ get area (29.6.3.4.2) and put area (29.6.3.4.3) pointers after initializing, moving from, or assigning to `buf` accordingly.

### 29.8.2.2 Constructors

#### [stringbuf.cons]

**explicit basic_stringbuf(ios_base::openmode which);**

1. **Effects**: Initializes the base class with `basic_streambuf()` (29.6.3.2), and `mode` with `which`. It is implementation-defined whether the sequence pointers (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) are initialized to null pointers.
2. **Postconditions**: `str().empty()` is true.

**explicit basic_stringbuf(  
    const basic_string<CharT, traits, Allocator>& s,  
    ios_base::openmode which = ios_base::in | ios_base::out);**

1. **Effects**: Initializes the base class with `basic_streambuf()` (29.6.3.2), `mode` with `which`, and `buf` with `s`, then calls `init_buf_ptrs()`.
2. **Effects**: Initializes the base class with `basic_streambuf()` (29.6.3.2), `mode` with `which`, and `buf` with `s`, then calls `init_buf_ptrs()`.
3. **Postconditions**: `str().empty()` is true.

**explicit basic_stringbuf(  
    basic_string<CharT, traits, Allocator>&& s,  
    ios_base::openmode which = ios_base::in | ios_base::out);**

1. **Effects**: Initializes the base class with `basic_streambuf()` (29.6.3.2), `mode` with `which`, and `buf` with `{s.a}`, then calls `init_buf_ptrs()`.
2. **Effects**: Initializes the base class with `basic_streambuf()` (29.6.3.2), `mode` with `which`, and `buf` with `s`, then calls `init_buf_ptrs()`.

### § 29.8.2.2 1429
basic_stringbuf(basic_stringbuf&& rhs, const Allocator& a);

10 Effects: Copy constructs the base class from rhs and initializes mode with rhs.mode. In the first form buf is initialized from std::move(rhs).str(). In the second form buf is initialized from {std::move(rhs).str(), a}. It is implementation-defined whether the sequence pointers in *this (eback(), gptr(), egptr(), pbase(), pptr(), epptr()) obtain the values which rhs had.

11 Postconditions: Let rhs_p refer to the state of rhs just prior to this construction and let rhs_a refer to the state of rhs just after this construction.

(11.1) — str() == rhs_p.str()
(11.2) — gptr() - eback() == rhs_p.gptr() - rhs_p.eback()
(11.3) — egptr() - eback() == rhs_p.egptr() - rhs_p.eback()
(11.4) — pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()
(11.5) — epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()
(11.6) — if (eback()) eback() != rhs_a.eback()
(11.7) — if (gptr()) gptr() != rhs_a.gptr()
(11.8) — if (egptr()) egptr() != rhs_a.egptr()
(11.9) — if (pbase()) pbase() != rhs_a.pbase()
(11.10) — if (pptr()) pptr() != rhs_a.pptr()
(11.11) — if (epptr()) epptr() != rhs_a.epptr()
(11.12) — getloc() == rhs_p.getloc()
(11.13) — rhs is empty but usable, as if std::move(rhs).str() was called.

29.8.2.3 Assignment and swap [stringbuf.assign]

basic_stringbuf& operator=(basic_stringbuf&& rhs);

1 Effects: After the move assignment *this has the observable state it would have had if it had been move constructed from rhs (see 29.8.2.2).

2 Returns: *this.

void swap(basic_stringbuf& rhs) noexcept(see below);

3 Preconditions: allocator_traits<Allocator>::propagate_on_container_swap::value is true or get_allocator() == s.get_allocator() is true.

4 Effects: Exchanges the state of *this and rhs.

5 Remarks: The expression inside noexcept is equivalent to:

allocator_traits<Allocator>::propagate_on_container_swap::value || allocator_traits<Allocator>::is_always_equal::value.

template<class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
basic_stringbuf<charT, traits, Allocator>& y) noexcept(noexcept(x.swap(y)));

6 Effects: Equivalent to: x.swap(y).

29.8.2.4 Member functions [stringbuf.members]

1 The member functions getting the underlying character sequence all refer to a high_mark value, where high_mark represents the position one past the highest initialized character in the buffer. Characters can be initialized by writing to the stream, by constructing the basic_stringbuf passing a basic_string argument, or by calling one of the str member functions passing a basic_string as an argument. In the latter case, all characters initialized prior to the call are now considered uninitialized (except for those characters re-initialized by the new basic_string).

void init_buf_ptrs(); // exposition only

2 Effects: Initializes the input and output sequences from buf according to mode.

3 Postconditions:
If `ios_base::out` is set in mode, `pbase()` points to `buf.front()` and `epptr() >= pbase() + buf.size()` is true; in addition, if `ios_base::ate` is set in mode, `pptr() == pbase() + buf.size()` is true, otherwise `pptr() == pbase()` is true.

If `ios_base::in` is set in mode, `eback()` points to `buf.front()`, and `(gptr() == eback() && egptr() == eback() + buf.size())` is true.

[Note 1: For efficiency reasons, stream buffer operations might violate invariants of `buf` while it is held encapsulated in the `basic_stringbuf`, e.g., by writing to characters in the range `[buf.data() + buf.size(), buf.data() + buf.capacity())`. All operations retrieving a `basic_string` from `buf` ensure that the `basic_string` invariants hold on the returned value. — end note]

Returns: `buf.get_allocator()`.

`basic_string<charT, traits, Allocator> str() const &;`  
Effects: Equivalent to:

`return basic_string<charT, traits, Allocator>(view(), get_allocator());`

```cpp
template<class SAlloc>
basic_string<charT, traits, SAlloc> str(const SAlloc& sa) const;
```

Constraints: `SAlloc` is a type that qualifies as an allocator (22.2.1).

Effects: Equivalent to:

`return basic_string<charT, traits, SAlloc>(view(), sa);`

`basic_string<charT, traits, Allocator> str() &&;`

Postconditions: The underlying character sequence `buf` is empty and `pbase()`, `pptr()`, `epptr()`, `eback()`, `gptr()`, and `egptr()` are initialized as if by calling `init_buf_ptrs()` with an empty `buf`.

Returns: A `basic_string<charT, traits, Allocator>` object move constructed from the `basic_stringbuf`'s underlying character sequence in `buf`. This can be achieved by first adjusting `buf` to have the same content as `view()`.

`basic_string_view<charT, traits> view() const noexcept;`

Let `sv` be `basic_string_view<charT, traits>`.  

Returns: A `sv` object referring to the `basic_stringbuf`'s underlying character sequence in `buf`:

(12.1) If `ios_base::out` is set in mode, then `sv(pbase(), high_mark-pbase())` is returned.

(12.2) Otherwise, if `ios_base::in` is set in mode, then `sv(eback(), egptr()-eback())` is returned.

(12.3) Otherwise, `sv()` is returned.

[Note 2: Using the returned `sv` object after destruction or invalidation of the character sequence underlying *this is undefined behavior, unless `sv.empty()` is true. — end note]

`void str(const basic_string<charT, traits, Allocator>& s);`

Effects: Equivalent to:

`buf = s;`

`init_buf_ptrs();`

```cpp
template<class SAlloc>
void str(const basic_string<charT, traits, SAlloc>& s);
```

Constraints: `is_same_v<SAlloc, Allocator>` is `false`.

Effects: Equivalent to:

`buf = s;`

`init_buf_ptrs();`

`void str(basic_string<charT, traits, Allocator>&& s);`

Effects: Equivalent to:
buffer = std::move(s);
init_buf_ptrs();

29.8.2.5 Overridden virtual functions

int_type underflow() override;

Returns: If the input sequence has a read position available, returns traits::to_int_type(*gptr()). Otherwise, returns traits::eof(). Any character in the underlying buffer which has been initialized is considered to be part of the input sequence.

int_type pbackfail(int_type c = traits::eof()) override;

Effects: Puts back the character designated by c to the input sequence, if possible, in one of three ways:

(2.1) If traits::eq_int_type(c, traits::eof()) returns false and if the input sequence has a putback position available, and if traits::eq(to_char_type(c), gptr()[-1]) returns true, assigns gptr() - 1 to gptr().
Returns: c.

(2.2) If traits::eq_int_type(c, traits::eof()) returns false and if the input sequence has a putback position available, and if mode & ios_base::out is nonzero, assigns c to *--gptr().
Returns: c.

(2.3) If traits::eq_int_type(c, traits::eof()) returns true and if the input sequence has a putback position available, assigns gptr() - 1 to gptr().
Returns: traits::not_eof(c).

Returns: As specified above, or traits::eof() to indicate failure.

Remarks: If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

int_type overflow(int_type c = traits::eof()) override;

Effects: Appends the character designated by c to the output sequence, if possible, in one of two ways:

(5.1) If traits::eq_int_type(c, traits::eof()) returns false and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls sputc(c).
Signals success by returning c.

(5.2) If traits::eq_int_type(c, traits::eof()) returns true, there is no character to append.
Signals success by returning a value other than traits::eof().

Returns: As specified above, or traits::eof() to indicate failure.

Remarks: The function can alter the number of write positions available as a result of any call.

The function can make a write position available only if ios_base::out is set in mode. To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus at least one additional write position. If ios_base::in is set in mode, the function alters the read end pointer egptr() to point just past the new write position.

pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out) override;

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 124.

For a sequence to be positioned, the function determines newoff as indicated in Table 125. If the sequence’s next pointer (either gptr() or pptr()) is a null pointer and newoff is nonzero, the positioning operation fails.

If (newoff + off) < 0, or if newoff + off refers to an uninitialized character (29.8.2.4), the positioning operation fails. Otherwise, the function assigns xbeg + newoff + off to the next pointer xnext.
Table 124: seekoff positioning [tab:stringbuf.seekoff.pos]

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ios_base::in</code> is set in which</td>
<td>positions the input sequence</td>
</tr>
<tr>
<td><code>ios_base::out</code> is set in which</td>
<td>positions the output sequence</td>
</tr>
<tr>
<td><code>both ios_base::in and ios_base::out</code> are set in which and either</td>
<td>positions both the input and the output sequences</td>
</tr>
<tr>
<td><code>way == ios_base::beg</code> or</td>
<td></td>
</tr>
<tr>
<td><code>way == ios_base::end</code></td>
<td></td>
</tr>
<tr>
<td>Otherwise</td>
<td>the positioning operation fails.</td>
</tr>
</tbody>
</table>

Table 125: newoff values [tab:stringbuf.seekoff.newoff]

<table>
<thead>
<tr>
<th>Condition</th>
<th>newoff Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>way == ios_base::beg</code></td>
<td>0</td>
</tr>
<tr>
<td><code>way == ios_base::cur</code></td>
<td><code>xnext - xbeg</code></td>
</tr>
<tr>
<td><code>way == ios_base::end</code></td>
<td><code>high_mark - xbeg</code></td>
</tr>
</tbody>
</table>

12 Returns: `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

13 `pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out) override;`

14 Effects: Equivalent to `seekoff(off_type(sp), ios_base::beg, which)`.

15 Returns: `sp` to indicate success, or `pos_type(off_type(-1))` to indicate failure.

16 `basic_streambuf<charT, traits>* setbuf(charT* s, streamsize n);

15 Effects: implementation-defined, except that `setbuf(0, 0)` has no effect.

16 Returns: `this`.

29.8.3 Class template basic_istringstream [istringstream]

29.8.3.1 General [istringstream.general]

namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_istringstream : public basic_istream<charT, traits> {
    public:
      using char_type = charT;
      using int_type = typename traits::int_type;
      using pos_type = typename traits::pos_type;
      using off_type = typename traits::off_type;
      using traits_type = traits;
      using allocator_type = Allocator;

      // 29.8.3.2, constructors
      basic_istringstream() : basic_istringstream(ios_base::in) {}
      explicit basic_istringstream(ios_base::openmode which);
      explicit basic_istringstream(
        const basic_string<charT, traits, Allocator>& s,
        ios_base::openmode which = ios_base::in);
      basic_istringstream(ios_base::openmode which, const Allocator& a);
    }
}
The class `basic_istringstream<charT, traits, Allocator>` supports reading objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the stringbuf object.

### 29.8.2.2 Constructors

```cpp
explicit basic_istringstream(Char_traits& traits, Allocator& s,
   ios_base::openmode which = ios_base::in);
``` 

**Effects:** Initializes the base class with `basic_istream<charT, traits>(addressof(sb))` (29.7.4.2) and `sb` with `basic_stringbuf<charT, traits, Allocator>(traits, Allocator)(s, which | ios_base::in)` (29.8.2.2).
basic_istringstream(ios_base::openmode which, const Allocator& a);

Effects: Initializes the base class with basic_istream<char, traits>(addressof(sb)) (29.7.4.2) and sb with basic_stringbuf<char, traits, Allocator>(which | ios_base::in, a) (29.8.2.2).

explicit basic_istringstream(
    basic_string<char, traits, Allocator>&& s,
    ios_base::openmode which = ios_base::in);

Effects: Initializes the base class with basic_istream<char, traits>(addressof(sb)) (29.7.4.2) and sb with basic_stringbuf<char, traits, Allocator>(std::move(s), which | ios_base::in) (29.8.2.2).

template<class SAlloc>
basic_istringstream(
    const basic_string<char, traits, SAlloc>& s,
    ios_base::openmode which, const Allocator& a);

Effects: Initializes the base class with basic_istream<char, traits>(addressof(sb)) (29.7.4.2) and sb with basic_stringbuf<char, traits, Allocator>(s, which | ios_base::in, a) (29.8.2.2).

template<class SAlloc>
explicit basic_istringstream(
    const basic_string<char, traits, SAlloc>& s,
    ios_base::openmode which = ios_base::in);

Effects: Initializes the base class with basic_istream<char, traits>(addressof(sb)) (29.7.4.2) and sb with basic_stringbuf<char, traits, Allocator>(s, which | ios_base::in) (29.8.2.2).

basic_istringstream(basic_istringstream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_stringbuf. Then calls basic_istream<char, traits>::set_rdbuf(addressof(sb)) to install the contained basic_stringbuf.

29.8.3.3 Assignment and swap

void swap(basic_istringstream& rhs);

Effects: Equivalent to:

    basic_istream<char, traits>::swap(rhs);
    sb.swap(rhs.sb);

template<class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x,
    basic_istringstream<charT, traits, Allocator>& y);

Effects: Equivalent to: x.swap(y).

29.8.3.4 Member functions

basic_stringbuf<char, traits, Allocator>* rdbuf() const;

Returns: const_cast<basic_stringbuf<char, traits, Allocator>>*(addressof(sb)).

basic_string<char, traits, Allocator> str() const &;

Effects: Equivalent to: return rdbuf()->str();

template<class SAlloc>
basic_string<char,Traits, SAlloc> str(const SAlloc& sa) const;

Effects: Equivalent to: return rdbuf()->str(sa);

basic_string<char,traits,Allocator> str() &&;

Effects: Equivalent to: return std::move(*rdbuf()).str();

basic_string_view<char, traits> view() const noexcept;

Effects: Equivalent to: return rdbuf()->view();

§ 29.8.3.4
void str(const basic_string<charT, traits, Allocator>& s);

Effects: Equivalent to: rdbuf() -> str(s);

template<class SAlloc>
void str(const basic_string<charT, traits, SAlloc>& s);

Effects: Equivalent to: rdbuf() -> str(s);

void str(basic_string<charT, traits, Allocator>&& s);

Effects: Equivalent to: rdbuf() -> str(std::move(s));

29.8.4 Class template basic_ostringstream

29.8.4.1 General

namespace std {
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_ostringstream : public basic_ostream<charT, traits> {
        public:
            using char_type = charT;
            using int_type = typename traits::int_type;
            using pos_type = typename traits::pos_type;
            using off_type = typename traits::off_type;
            using traits_type = traits;
            using allocator_type = Allocator;

            // 29.8.4.2, constructors
            basic_ostringstream() : basic_ostringstream(ios_base::out) {}
            explicit basic_ostringstream(ios_base::openmode which);
            explicit basic_ostringstream(
                const basic_string<charT, traits, Allocator>&& s,
                ios_base::openmode which = ios_base::out);
            basic_ostringstream(ios_base::openmode which, const Allocator& a);
            explicit basic_ostringstream(
                const basic_string<charT, traits, Allocator>& s,
                ios_base::openmode which = ios_base::out);
            template<class SAlloc>
            basic_ostringstream(
                const basic_string<charT, traits, SAlloc>& s, const Allocator& a)
                : basic_ostringstream(s, ios_base::out, a) {}
            template<class SAlloc>
            basic_ostringstream(
            const basic_string<charT, traits, SAlloc>& s,
            ios_base::openmode which, const Allocator& a);
            template<class SAlloc>
            explicit basic_ostringstream(
            const basic_string<charT, traits, SAlloc>& s,
            ios_base::openmode which = ios_base::out);
            basic_ostringstream(const basic_ostringstream&) = delete;
            basic_ostringstream(basic_ostringstream&& rhs);

            // 29.8.4.3, assign and swap
            basic_ostringstream& operator=(const basic_ostringstream&) = delete;
            basic_ostringstream& operator=(basic_ostringstream&& rhs);
            void swap(basic_ostringstream& rhs);

            // 29.8.4.4, members
            basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
            basic_string<charT, traits, Allocator> str() const &;
            template<class SAlloc>
            basic_string<charT, traits, SAlloc> str(const SAlloc& sa) const;
            basic_string<charT, traits, Allocator> str() &&;
            basic_string_view<charT, traits> view() const noexcept;
        }
void str(const basic_string<charT, traits, Allocator>& s);
template<class SAlloc>
void str(const basic_string<charT, traits, SAlloc>& s);
void str(basic_string<charT, traits, Allocator>&& s);

private:
  basic_stringbuf<charT, traits, Allocator> sb;  // exposition only
};

template<class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,
           basic_ostringstream<charT, traits, Allocator>& y);

§ 29.8.4.2 1437

The class basic_ostringstream supports writing objects of class basic_string. It uses a basic_stringbuf object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

(1.1) — sb, the stringbuf object.

29.8.4.2 Constructors

explicit basic_ostringstream(ios_base::openmode which);

1 Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.2) and sb with basic_stringbuf<charT, traits, Allocator>(which | ios_base::out) (29.8.2.2).

explicit basic_ostringstream(
  const basic_string<charT, traits, Allocator>& s,
  ios_base::openmode which = ios_base::out);

2 Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.2) and sb with basic_stringbuf<charT, traits, Allocator>(s, which | ios_base::out) (29.8.2.2).

basic_ostringstream(ios_base::openmode which, const Allocator& a);

3 Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.2) and sb with basic_stringbuf<charT, traits, Allocator>(which | ios_base::out, a) (29.8.2.2).

explicit basic_ostringstream(
  basic_string<charT, traits, Allocator>&& s,
  ios_base::openmode which = ios_base::out);

4 Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.2) and sb with basic_stringbuf<charT, traits, Allocator>(std::move(s), which | ios_base::out) (29.8.2.2).

template<class SAlloc>
basic_ostringstream(  
  const basic_string<charT, traits, SAlloc>& s,
  ios_base::openmode which, const Allocator& a);

5 Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.2) and sb with basic_stringbuf<charT, traits, Allocator>(s, which | ios_base::out, a) (29.8.2.2).

template<class SAlloc>
explicit basic_ostringstream(  
  const basic_string<charT, traits, SAlloc>&& s,
  ios_base::openmode which = ios_base::out);

6 Constraints: is_same_v<SAlloc, Allocator> is false.

7 Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.2) and sb with basic_stringbuf<charT, traits, Allocator>(s, which | ios_base::out) (29.8.2.2).
basic_ostringstream(basic_ostringstream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_stringbuf. Then calls basic_ostream<charT, traits>::set_rdbuf(addressof(sb)) to install the contained basic_stringbuf.

29.8.4.3 Assignment and swap

void swap(basic_ostringstream& rhs);

Effects: Equivalent to:

basic_ostream<charT, traits>::swap(rhs);
sb.swap(rhs.sb);

template<class charT, class traits, class Allocator>

void swap(basic_ostringstream<charT, traits, Allocator>& x,
          basic_ostringstream<charT, traits, Allocator>& y);

Effects: Equivalent to: x.swap(y).

29.8.4.4 Member functions

basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

Returns: const_cast<basic_stringbuf<charT, traits, Allocator>*>(addressof(sb)).

basic_string<charT, traits, Allocator> str() const &;

Effects: Equivalent to: return rdbuf()->str();

template<class SAloc>

basic_string<charT,traits,SAloc> str(const SAloc& sa) const;

Effects: Equivalent to: return rdbuf()->str(sa);

basic_string<charT,traits,Allocator> str() &&;

Effects: Equivalent to: return std::move(*rdbuf()).str();

basic_string_view<charT, traits> view() const noexcept;

Effects: Equivalent to: return rdbuf()->view();

void str(const basic_string<charT, traits, Allocator>& s);

Effects: Equivalent to: rdbuf()->str(s);

template<class SAloc>

void str(const basic_string<charT, traits, SAloc>& s);

Effects: Equivalent to: rdbuf()->str(s);

void str(basic_string<charT, traits, Allocator>&& s);

Effects: Equivalent to: rdbuf()->str(std::move(s));

29.8.5 Class template basic_stringstream

29.8.5.1 General

namespace std {

template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT>>

class basic_stringstream : public basic_iostream<charT, traits> {
public:
    using char_type = charT;
    using int_type = typename traits::int_type;
    using pos_type = typename traits::pos_type;
    using off_type = typename traits::off_type;
    using traits_type = traits;
    using allocator_type = Allocator;

§ 29.8.5.1
The class template `basic_stringstream<charT, traits>` supports reading and writing from objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as

(1.1) — `sb`, the stringbuf object.

### 29.8.5.2 Constructors

**[stringstream.cons]**

```cpp
explicit basic_stringstream(ios_base::openmode which);
```

**Effects:** Initializes the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.7.2)
and sb with basic_stringbuf<charT, traits, Allocator>(which).

explicit basic_stringstream(
    const basic_string<charT, traits, Allocator>& s,
    ios_base::openmode which = ios_base::out | ios_base::in);

2 Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.7.2) and sb with basic_stringbuf<charT, traits, Allocator>(s, which).

basic_stringstream(ios_base::openmode which, const Allocator& a);

3 Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.7.2) and sb with basic_stringbuf<charT, traits, Allocator>(which, a) (29.8.2.2).

explicit basic_stringstream(
    basic_string<charT, traits, Allocator>&& s,
    ios_base::openmode which = ios_base::out | ios_base::in);

4 Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.7.2) and sb with basic_stringbuf<charT, traits, Allocator>(std::move(s), which) (29.8.2.2).

template<class SAlloc>
    basic_stringstream(
        const basic_string<charT, traits, SAlloc>& s,
        ios_base::openmode which, const Allocator& a);

5 Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.7.2) and sb with basic_stringbuf<charT, traits, Allocator>(s, which, a) (29.8.2.2).

template<class SAlloc>
    explicit basic_stringstream(
        const basic_string<charT, traits, SAlloc>& s,
        ios_base::openmode which = ios_base::out | ios_base::in);

6 Constraints: is_same_v<SAlloc, Allocator> is false.

7 Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.7.2) and sb with basic_stringbuf<charT, traits, Allocator>(s, which) (29.8.2.2).

basic_stringstream(basic_stringstream&& rhs);

8 Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_stringbuf. Then calls basic_istream<charT, traits>::set_rdbuf(addressof(sb)) to install the contained basic_stringbuf.

29.8.5.3 Assignment and swap [stringstream.assign]

void swap(basic_stringstream& rhs);

1 Effects: Equivalent to:
    basic_iostream<charT, traits>::swap(rhs);
    sb.swap(rhs.sb);

template<class charT, class traits, class Allocator>
    void swap(basic_stringstream<charT, traits, Allocator>& x,
    basic_stringstream<charT, traits, Allocator>& y);

2 Effects: Equivalent to: x.swap(y).

29.8.5.4 Member functions [stringstream.members]

    basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

1 Returns: const_cast<basic_stringbuf<charT, traits, Allocator>*>(addressof(sb)).

    basic_string<charT, traits, Allocator> str() const &;

2 Effects: Equivalent to: return rdbuf()->str();
template<class SAlloc>
  basic_string<charT,traits,SAlloc> str(const SAlloc& sa) const;

  Effects: Equivalent to: return rdbuf()->str(sa);

basic_string<charT,traits,Allocator> str() &&;

  Effects: Equivalent to: return std::move(*rdbuf()).str();

basic_string_view<charT,traits> view() const noexcept;

  Effects: Equivalent to: return rdbuf()->view();

void str(const basic_string<charT, traits, Allocator>& s);

  Effects: Equivalent to: rdbuf()->str(s);

template<class SAlloc>
  void str(const basic_string<charT, traits, SAlloc>& s);

  Effects: Equivalent to: rddbuf()->str(s);

void str(basic_string<charT, traits, Allocator>&& s);

  Effects: Equivalent to: rddbuf()->str(std::move(s));
The class `basic_filebuf<` charT`, traits>` associates both the input sequence and the output sequence with a file.

The restrictions on reading and writing a sequence controlled by an object of class `basic_filebuf<` charT`, traits>` are the same as for reading and writing with the C standard library `FILEs`.

In particular:

1. If the file is not open for reading the input sequence cannot be read.
2. If the file is not open for writing the output sequence cannot be written.
3. A joint file position is maintained for both the input sequence and the output sequence.
An instance of `basic_filebuf` behaves as described in 29.9.2 provided `traits::pos_type` is `fpos<traits::state_type>`. Otherwise the behavior is undefined.

In order to support file I/O and multibyte/wide character conversion, conversions are performed using members of a facet, referred to as `a_codecvt` in following subclauses, obtained as if by:

```cpp
const codecvt<charT, char, typename traits::state_type>& a_codecvt = use_facet<codecvt<charT, char, typename traits::state_type>>(getloc());
```

### 29.9.2.2 Constructors

#### basic_filebuf();

1. **Effects:** Initializes the base class with `basic_streambuf<charT, traits>()` (29.6.3.2).
2. **Postconditions:** `is_open() == false`.

#### basic_filebuf(basic_filebuf&& rhs);

3. **Effects:** It is implementation-defined whether the sequence pointers in `*this` (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) obtain the values which `rhs` had. Whether they do or not, `*this` and `rhs` reference separate buffers (if any at all) after the construction. Additionally `*this` references the file which `rhs` did before the construction, and `rhs` references no file after the construction. The openmode, locale and any other state of `rhs` is also copied.
4. **Postconditions:** Let `rhs_p` refer to the state of `rhs` just prior to this construction and let `rhs_a` refer to the state of `rhs` just after this construction.
   - `is_open() == rhs_p.is_open()`  
   - `rhs_a.is_open() == false`  
   - `gptr() - eback() == rhs_p.gptr() - rhs_p.eback()`  
   - `egptr() - eback() == rhs_p.egptr() - rhs_p.eback()`  
   - `pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()`  
   - `epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()`  
   - `if (eback()) eback() != rhs_a.eback()`  
   - `if (gptr()) gptr() != rhs_a.gptr()`  
   - `if (egptr()) egptr() != rhs_a.egptr()`  
   - `if (pbase()) pbase() != rhs_a.pbase()`  
   - `if (pptr()) pptr() != rhs_a.pptr()`  
   - `if (epptr()) epptr() != rhs_a.epptr()`
5. **Effects:** Calls `close()`. If an exception occurs during the destruction of the object, including the call to `close()`, the exception is caught but not rethrown (see 16.4.6.13).

### 29.9.2.3 Assignment and swap

#### basic_filebuf& operator=(basic_filebuf&& rhs);

1. **Effects:** Calls `close()` then move assigns from `rhs`. After the move assignment `*this` has the observable state it would have had if it had been move constructed from `rhs` (see 29.9.2.2).
2. **Returns:**`*this`.

#### void swap(basic_filebuf& rhs);

3. **Effects:** Exchanges the state of `*this` and `rhs`.

#### template<class charT, class traits>

```cpp
void swap(basic_filebuf<charT, traits>& x, basic_filebuf<charT, traits>& y);
```

4. **Effects:** Equivalent to: `x.swap(y)`. 

---

§ 29.9.2.3
29.9.2.4 Member functions

bool is_open() const;

Returns: true if a previous call to open succeeded (returned a non-null value) and there has been no intervening call to close.

basic_filebuf* open(const char* s, ios_base::openmode mode);
basic_filebuf* open(const filesystem::path::value_type* s, ios_base::openmode mode); // wide systems only; see 29.9.1

Preconditions: s points to a NTCTS (3.36).

Effects: If is_open() != false, returns a null pointer. Otherwise, initializes the filebuf as required. It then opens the file to which s resolves, if possible, as if by a call to fopen with the second argument determined from mode & ~ios_base::ate as indicated in Table 126. If mode is not some combination of flags shown in the table then the open fails.

Table 126: File open modes

<table>
<thead>
<tr>
<th>ios_base flag combination</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>out</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

If the open operation succeeds and ios_base::ate is set in mode, positions the file to the end (as if by calling fseek(file, 0, SEEK_END), where file is the pointer returned by calling fopen).326

If the repositioning operation fails, calls close() and returns a null pointer to indicate failure.

Returns: this if successful, a null pointer otherwise.

basic_filebuf* open(const string& s, ios_base::openmode mode);
basic_filebuf* open(const filesystem::path& s, ios_base::openmode mode);

Returns: open(s.c_str(), mode);

basic_filebuf* close();

Effects: If is_open() == false, returns a null pointer. If a put area exists, calls overflow(traits::eof()) to flush characters. If the last virtual member function called on *this (between underflow, overflow, seekoff, and seekpos) was overflow then calls a_codecvt.unshift (possibly several times) to determine a termination sequence, inserts those characters and calls overflow(traits::eof()) again. Finally, regardless of whether any of the preceding calls fails or throws an exception, the function closes the file (as if by calling fclose(file)). If any of the calls made by the function,

326 The macro SEEK_END is defined, and the function signatures fopen(const char*, const char*) and fseek(FILE*, long, int) are declared, in <cstdio> (29.12.1).
including `fclose`, fails, `close` fails by returning a null pointer. If one of these calls throws an exception, the exception is caught and rethrown after closing the file.

**Postconditions:** `is_open() == false`.

**Returns:** `this` on success, a null pointer otherwise.

### 29.9.2.5 Overridden virtual functions

**streamsize showmanyc() override;**

**Effects:** Behaves the same as `basic_streambuf::showmanyc()` (29.6.3.5).

**Remarks:** An implementation may provide an overriding definition for this function signature if it can determine whether more characters can be read from the input sequence.

**int_type underflow() override;**

**Effects:** Behaves according to the description of `basic_streambuf<charT, traits>::underflow()`, with the specialization that a sequence of characters is read from the input sequence as if by reading from the associated file into an internal buffer (`extern_buf`) and then as if by doing:

```
codecvt_base::result r = a_codecvt.in(state, extern_buf, extern_buf+XSIZE, extern_end, intern_buf, intern_buf+ISIZE, intern_end);
```

This shall be done in such a way that the class can recover the position (`fpos_t`) corresponding to each character between `intern_buf` and `intern_end`. If the value of `r` indicates that `a_codecvt.in()` ran out of space in `intern_buf`, retry with a larger `intern_buf`.

**int_type uflow() override;**

**Effects:** Behaves according to the description of `basic_streambuf<charT, traits>::uflow()`, with the specialization that a sequence of characters is read from the input sequence as if by reading from the associated file into an internal buffer (`extern_buf`) and then as if by doing:

**int_type pbackfail(int_type c = traits::eof()) override;**

**Effects:** Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

1. **(5.1)** If `traits::eq_int_type(c, traits::eof())` returns `false` and if the function makes a putback position available and if `traits::eq(to_char_type(c), gptr()[−1])` returns `true`, decrements the next pointer for the input sequence, `gptr()`.

   Returns: `c`.

2. **(5.2)** If `traits::eq_int_type(c, traits::eof())` returns `false` and if the function makes a putback position available and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores `c` there.

   Returns: `c`.

3. **(5.3)** If `traits::eq_int_type(c, traits::eof())` returns `true`, and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, `gptr()`.

   Returns: `traits::not_eof(c)`.

**Returns:** As specified above, or `traits::eof()` to indicate failure.

**Remarks:** If `is_open() == false`, the function always fails.

The function does not put back a character directly to the input sequence.

If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.
int_type overflow(int_type c = traits::eof()) override;

Effects: Behaves according to the description of basic_streambuf<charT, traits>::overflow(c), except that the behavior of “consuming characters” is performed by first converting as if by:

```c
charT* b = pbase();
charT* p = pptr();
char* end;
char xbuf[XSIZE];
char* xbuf_end;
codecvt_base::result r =
a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);

and then
```

1. If \( r == codecvt_base::error \) then fail.
2. If \( r == codecvt_base::noconv \) then output characters from \( b \) up to (and not including) \( p \).
3. Otherwise output from \( xbuf \) to \( xbuf_end \), and fail if output fails. At this point if \( b != p \) and \( b == end \) (\( xbuf \) isn’t large enough) then increase \( XSIZE \) and repeat from the beginning.

Returns: traits::not_eof(c) to indicate success, and traits::eof() to indicate failure. If \!is_open() == false, the function always fails.

basic_streambuf* setbuf(char_type* s, streamsize n) override;

Effects: If setbuf(0, 0) is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined. “Unbuffered” means that \( pbase() \) and \( pptr() \) always return null and output to the file should appear as soon as possible.

pos_type seekoff(off_type off, ios_base::seekdir way,

ios_base::openmode which

= ios_base::in | ios_base::out) override;

Effects: Let width denote a_codecvt.encoding(). If \!is_open() == false, or \( off != 0 \) && width \( <= 0 \), then the positioning operation fails. Otherwise, if \( way != basic_ios::cur \) or \( off != 0 \), and if the last operation was output, then update the output sequence and write any unshift sequence. Next, seek to the new position: if \( width > 0 \), call fseek(file, width * off, whence), otherwise call fseek(file, 0, whence).

Returns: A newly constructed pos_type object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns pos_type(off_type(-1)).

Remarks: “The last operation was output” means either the last virtual operation was overflow or the put buffer is non-empty. “Write any unshift sequence” means, if \( width \) if less than zero then call a_codecvt.unshift(state, xbuf, xbuf+XSIZE, xbuf_end) and output the resulting unshift sequence. The function determines one of three values for the argument whence, of type int, as indicated in Table 127.

Table 127: seekoff effects

<table>
<thead>
<tr>
<th>way</th>
<th>stdio Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic_ios::beg</td>
<td>SEEK_SET</td>
</tr>
<tr>
<td>basic_ios::cur</td>
<td>SEEK_CUR</td>
</tr>
<tr>
<td>basic_ios::end</td>
<td>SEEK_END</td>
</tr>
</tbody>
</table>

pos_type seekpos(pos_type sp,

ios_base::openmode which

§ 29.9.2.5
Alters the file position, if possible, to correspond to the position stored in \textit{sp} (as described below). Altering the file position performs as follows:

1. if \((\text{om} \& \text{ios\_base\::\:out}) \neq 0\), then update the output sequence and write any unshift sequence;
2. set the file position to \textit{sp} as if by a call to \texttt{fsetpos};
3. if \((\text{om} \& \text{ios\_base\::\:in}) \neq 0\), then update the input sequence;

where \textit{om} is the open mode passed to the last call to \texttt{open()}. The operation fails if \textit{is\_open()} returns false.

If \textit{sp} is an invalid stream position, or if the function positions neither sequence, the positioning operation fails. If \textit{sp} has not been obtained by a previous successful call to one of the positioning functions (\texttt{seekoff} or \texttt{seekpos}) on the same file the effects are undefined.

\textit{Returns:} \textit{sp} on success. Otherwise returns \texttt{pos\_type(off\_type(-1))}.

\begin{verbatim}
int sync() override;
\end{verbatim}

\textit{Effects:} If a put area exists, calls \texttt{filebuf\::overflow} to write the characters to the file, then flushes the file as if by calling \texttt{fflush(file)}. If a get area exists, the effect is implementation-defined.

\begin{verbatim}
void imbue(const locale& loc) override;
\end{verbatim}

\textit{Preconditions:} If the file is not positioned at its beginning and the encoding of the current locale as determined by \texttt{a\_codecvt\::\encoding()} is state-dependent (28.4.2.5.3) then that facet is the same as the corresponding facet of \textit{loc}.

\textit{Effects:} Causes characters inserted or extracted after this call to be converted according to \textit{loc} until another call of \texttt{imbue}.

\textit{Remarks:} This may require reconversion of previously converted characters. This in turn may require the implementation to be able to reconstruct the original contents of the file.

### 29.9.3 Class template basic\_ifstream

#### 29.9.3.1 General

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_ifstream : public basic_istream<charT, traits> {
        public:
            using char_type = charT;
            using int_type = typename traits::int_type;
            using pos_type = typename traits::pos_type;
            using off_type = typename traits::off_type;
            using traits_type = traits;

            // 29.9.3.2, constructors
            basic_ifstream();
            explicit basic_ifstream(const char* s,
                ios_base::openmode mode = ios_base::in);
            explicit basic_ifstream(const filesystem::path::value_type* s,
                ios_base::openmode mode = ios_base::in); // wide systems only; see 29.9.1
            explicit basic_ifstream(const string& s,
                ios_base::openmode mode = ios_base::in);
            explicit basic_ifstream(const filesystem::path& s,
                ios_base::openmode mode = ios_base::in);
            basic_ifstream(const basic_ifstream&) = delete;
            basic_ifstream(basic_ifstream&& rhs);

            // 29.9.3.3, assign and swap
            basic_ifstream& operator=(const basic_ifstream&) = delete;
            basic_ifstream& operator=(basic_ifstream&& rhs);
            void swap(basic_ifstream& rhs);
        }
    }
}```
1 The class \texttt{basic_ifstream<\texttt{charT}, \texttt{traits}>} supports reading from named files. It uses a \texttt{basic_filebuf<\texttt{charT}, \texttt{traits}>} object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

\begin{equation}
\text{(1.1) \hspace{1cm} \texttt{sb}, the \texttt{filebuf} object.}
\end{equation}

29.9.3.2 Constructors \hfill [ifstream.cons]

\begin{itemize}
\item \texttt{basic_ifstream();} \\
\textbf{Effects:} Initializes the base class with \texttt{basic_istream<\texttt{charT}, \texttt{traits}>}(\texttt{addressof(sb)}) (29.7.4.2.2) and \texttt{sb} with \texttt{basic_filebuf<\texttt{charT}, \texttt{traits}>}() (29.9.2.2).

\item \texttt{explicit basic_ifstream(const char* s, \texttt{ios_base::openmode mode = \texttt{ios_base::in});}} \\
\texttt{explicit basic_ifstream(const filesystem::path::value_type* s, \texttt{ios_base::openmode mode = \texttt{ios_base::in}); // wide systems only; see 29.9.1}} \\
\texttt{explicit basic_ifstream(const string& s, \texttt{ios_base::openmode mode = \texttt{ios_base::in});}} \\
\texttt{explicit basic_ifstream(const filesystem::path& s, \texttt{ios_base::openmode mode = \texttt{ios_base::in});}} \\
\texttt{void close();} \\
\texttt{private:} \\
\texttt{basic_filebuf<\texttt{charT}, \texttt{traits}> sb; // exposition only} \\
\end{itemize}

29.9.3.3 Assignment and swap \hfill [ifstream.assign]

\begin{itemize}
\item \texttt{void swap(basic_ifstream& rhs);} \\
\textbf{Effects:} Exchanges the state of \texttt{*this} and \texttt{rhs} by calling \texttt{basic_istream<\texttt{charT}, \texttt{traits}>::swap(rhs)} and \texttt{sb.swap(rhs.sb)}. \\
\item \texttt{template<class charT, class traits> \hspace{1cm} void swap(basic_ifstream<\texttt{charT}, \texttt{traits}>& x, basic_ifstream<\texttt{charT}, \texttt{traits}>& y);} \\
\textbf{Effects:} Equivalent to: \texttt{x.swap(y).}
\end{itemize}
29.9.3.4 Member functions

basic_filebuf<charT, traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT, traits>*>(addressof(sb)).

bool is_open() const;

Returns: rdbuf()->is_open().

void open(const char* s, ios_base::openmode mode = ios_base::in);
void open(const filesystem::path::value_type* s,
           ios_base::openmode mode = ios_base::in); // wide systems only; see 29.9.1

Effects: Calls rdbuf()->open(s, mode | ios_base::in). If that function does not return a null pointer calls clear(), otherwise calls setstate(failbit) (which may throw ios_base::failure) (29.5.5.4).

void open(const string& s, ios_base::openmode mode = ios_base::in);
void open(const filesystem::path& s, ios_base::openmode mode = ios_base::in);

Effects: Calls open(s.c_str(), mode).

void close();

Effects: Calls rdbuf()->close() and, if that function returns a null pointer, calls setstate(failbit) (which may throw ios_base::failure) (29.5.5.4).

29.9.4 Class template basic_ofstream

29.9.4.1 General

namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_ofstream : public basic_ostream<charT, traits> {
        public:
            using char_type = charT;
            using int_type = typename traits::int_type;
            using pos_type = typename traits::pos_type;
            using off_type = typename traits::off_type;
            using traits_type = traits;

            // 29.9.4.2, constructors
            basic_ofstream();
            explicit basic_ofstream(const char* s,
                                       ios_base::openmode mode = ios_base::out);
            explicit basic_ofstream(const filesystem::path::value_type* s,
                                       // wide systems only; see 29.9.1
                                       ios_base::openmode mode = ios_base::out);
            explicit basic_ofstream(const string& s,
                                       ios_base::openmode mode = ios_base::out);
            explicit basic_ofstream(const filesystem::path& s,
                                       ios_base::openmode mode = ios_base::out);
            basic_ofstream(const basic_ofstream&) = delete;
            basic_ofstream(basic_ofstream&& rhs);

            // 29.9.4.3, assign and swap
            basic_ofstream& operator=(const basic_ofstream&);
            basic_ofstream& operator=(basic_ofstream&& rhs);
            void swap(basic_ofstream& rhs);

            // 29.9.4.4, members
            basic_filebuf<charT, traits>* rdbuf() const;

            bool is_open() const;
            void open(const char* s, ios_base::openmode mode = ios_base::out);
            void open(const filesystem::path::value_type* s,
                           ios_base::openmode mode = ios_base::out); // wide systems only; see 29.9.1
            void open(const string& s, ios_base::openmode mode = ios_base::out);
            void open(const filesystem::path& s, ios_base::openmode mode = ios_base::out);

§ 29.9.4.1 1449
The class `basic_ofstream<charT, traits>` supports writing to named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

```cpp
— sb, the filebuf object.
```

### 29.9.4.2 Constructors

```cpp
basic_ofstream();
```

**Effects**: Initializes the base class with `basic_ostream<charT, traits>(addressof(sb))` (29.7.5.2.2) and `sb` with `basic_filebuf<charT, traits>()` (29.9.2.2).

```cpp
explicit basic_ofstream(const char* s,
    ios_base::openmode mode = ios_base::out);
```

**Effects**: Initializes the base class with `basic_ostream<charT, traits>(addressof(sb))` (29.7.5.2.2) and `sb` with `basic_filebuf<charT, traits>()` (29.9.2.2), then calls `rdbuf()->open(s, mode | ios_base::out)`. If that function returns a null pointer, calls `setstate(failbit)`.

```cpp
explicit basic_ofstream(const string& s,
    ios_base::openmode mode = ios_base::out);
```

**Effects**: Equivalent to: `basic_ofstream(s.c_str(), mode)`.

```cpp
basic_ofstream(basic_ofstream&& rhs);
```

**Effects**: Move constructs the base class, and the contained `basic_filebuf`. Then calls `basic_ostream<charT, traits>::set_rdbuf(addressof(sb))` to install the contained `basic_filebuf`.

### 29.9.4.3 Assignment and swap

```cpp
void swap(basic_ofstream& rhs);
```

**Effects**: Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```cpp
template<class charT, class traits>
void swap(basic_ofstream<charT, traits>& x,
    basic_ofstream<charT, traits>& y);
```

**Effects**: Equivalent to: `x.swap(y)`.

### 29.9.4.4 Member functions

```cpp
basic_filebuf<charT, traits>* rdbuf() const;
```

**Returns**: `const_cast<basic_filebuf<charT, traits>*>(addressof(sb))`.

```cpp
bool is_open() const;
```

**Returns**: `rdbuf()->is_open()`.

```cpp
void open(const char* s, ios_base::openmode mode = ios_base::out);
```
void open(const filesystem::path::value_type* s, 
   ios_base::openmode mode = ios_base::out); // wide systems only; see 29.9.1

Effects: Calls rdbuf()->open(s, mode | ios_base::out). If that function does not return a null pointer calls clear(), otherwise calls setstate(failbit) (which may throw ios_base::failure) (29.5.5.4).

void close();

Effects: Calls rdbuf()->close() and, if that function fails (returns a null pointer), calls setstate(failbit) (which may throw ios_base::failure) (29.5.5.4).

void open(const string& s, ios_base::openmode mode = ios_base::out);
void open(const filesystem::path& s, ios_base::openmode mode = ios_base::out);

Effects: Calls open(s.c_str(), mode).

29.9.5 Class template basic_fstream

29.9.5.1 General

namespace std {
   template<class charT, class traits = char_traits<charT>>
   class basic_fstream : public basic_iostream<charT, traits> {
      public:
         using char_type = charT;
         using int_type = typename traits::int_type;
         using pos_type = typename traits::pos_type;
         using off_type = typename traits::off_type;
         using traits_type = traits;

         // 29.9.5.2, constructors
         basic_fstream();
         explicit basic_fstream(
            const char* s,
            ios_base::openmode mode = ios_base::in | ios_base::out);
         explicit basic_fstream(
            const filesystem::path::value_type* s,
            ios_base::openmode mode = ios_base::in|ios_base::out); // wide systems only; see 29.9.1
         explicit basic_fstream(
            const string& s,
            ios_base::openmode mode = ios_base::in | ios_base::out);
         explicit basic_fstream(
            const filesystem::path& s,
            ios_base::openmode mode = ios_base::in | ios_base::out);
         basic_fstream(const basic_fstream&) = delete;
         basic_fstream(basic_fstream&& rhs);

         // 29.9.5.3, assign and swap
         basic_fstream& operator=(const basic_fstream&) = delete;
         basic_fstream& operator=(basic_fstream&& rhs);
         void swap(basic_fstream& rhs);

         // 29.9.5.4, members
         basic_filebuf<charT, traits>* rdbuf() const;
         bool is_open() const;
         void open(
            const char* s,
            ios_base::openmode mode = ios_base::in | ios_base::out);
         void open(
            const filesystem::path::value_type* s,
            ios_base::openmode mode = ios_base::in|ios_base::out); // wide systems only; see 29.9.1
         void open(
            const string& s,
            ios_base::openmode mode = ios_base::in | ios_base::out);

§ 29.9.5.1
The class template `basic_fstream<charT, traits>` supports reading and writing from named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequences. For the sake of exposition, the maintained data is presented here as:

(1.1) — sb, the `basic_filebuf` object.

### 29.9.5.2 Constructors

```cpp
basic_fstream();
```

**Effects**: Initializes the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.7.2) and sb with `basic_filebuf<charT, traits>()`.

```cpp
explicit basic_fstream(
    const char* s,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

**Effects**: Initializes the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.7.2) and sb with `basic_filebuf<charT, traits>()`. Then calls `rdbuf()->open(s, mode)`. If that function returns a null pointer, calls `setstate(failbit)`.

```cpp
explicit basic_fstream(
    const string& s,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

**Effects**: Equivalent to: `basic_fstream(s.c_str(), mode)`.

```cpp
basic_fstream(basic_fstream&& rhs);
```

**Effects**: Move constructs the base class, and the contained `basic_filebuf`. Then calls `basic_iostream<charT, traits>::set_rdbuf(addressof(sb))` to install the contained `basic_filebuf`.

### 29.9.5.3 Assignment and swap

```cpp
void swap(basic_fstream& rhs);
```

**Effects**: Exchanges the state of *this and rhs by calling `basic_iostream<charT, traits>::swap(rhs)` and sb.swap(rhs.sb).

```cpp
template<class charT, class traits>
void swap(basic_fstream<charT, traits>& x,
          basic_fstream<charT, traits>& y);
```

**Effects**: Equivalent to: `x.swap(y)`.

### 29.9.5.4 Member functions

```cpp
basic_filebuf<charT, traits>* rdbuf() const;
```

**Returns**: `const_cast<basic_filebuf<charT, traits>*>(addressof(sb))`. 

§ 29.9.5.4
bool is_open() const;

Returns: rdbuf()->is_open().

void open(const char* s);
ios_base::openmode mode = ios_base::in | ios_base::out);
void open(const filesystem::path::value_type* s);
ios_base::openmode mode = ios_base::in | ios_base::out);  // wide systems only; see 29.9.1

Effects: Calls rdbuf()->open(s, mode). If that function does not return a null pointer calls clear()
otherwise calls setstate(failbit) (which may throw ios_base::failure) (29.5.5.4).

void open(const string& s);
ios_base::openmode mode = ios_base::in | ios_base::out);
void open(const filesystem::path& s);
ios_base::openmode mode = ios_base::in | ios_base::out);

Effects: Calls open(s.c_str(), mode).

void close();

Effects: Calls rdbuf()->close() and, if that function returns a null pointer, calls setstate(failbit)
(which may throw ios_base::failure) (29.5.5.4).

29.10  Synchronized output streams

29.10.1  Header <syncstream> synopsis

The header <syncstream> provides a mechanism to synchronize execution agents writing to the same stream.

29.10.2  Class template basic_syncbuf

namespace std {

template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
class basic_syncbuf {

template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
class osyncstream {

template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
class wosyncstream {

}
Class template `basic_syncbuf` stores character data written to it, known as the associated output, into internal buffers allocated using the object’s allocator. The associated output is transferred to the wrapped stream buffer object `*wrapped` when `emit()` is called or when the `basic_syncbuf` object is destroyed. Such transfers are atomic with respect to transfers by other `basic_syncbuf` objects with the same wrapped stream buffer object.

### 29.10.2.2 Construction and destruction

```cpp
basic_syncbuf() : basic_syncbuf(nullptr) {}  
explicit basic_syncbuf(streambuf_type* obuf) : basic_syncbuf(obuf, Allocator()) {}  
basic_syncbuf(streambuf_type*, const Allocator&);  
~basic_syncbuf();
```

**Effects**: Sets `wrapped` to `obuf`.

**Postconditions**: `get_wrapped() == obuf` and `get_allocator() == allocator` are true.

**Throws**: Nothing unless an exception is thrown by the construction of a mutex or by memory allocation.

**Remarks**: A copy of `allocator` is used to allocate memory for internal buffers holding the associated output.

```cpp
basic_syncbuf(streambuf_type* obuf, const Allocator& allocator);  
```

**Effects**: The value returned by `this->get_wrapped()` is the value returned by `other.get_wrapped()` prior to calling this constructor. Output stored in `other` prior to calling this constructor will be stored in `*this` afterwards. `other.rdbuf() -> pbase() == other.rdbuf() -> pptr()` and `other.get_wrapped() == nullptr` are true.

**Postconditions**: The value returned by `this->get_wrapped()` is the value returned by `other.get_wrapped()` prior to calling this constructor. Output stored in `other` prior to calling this constructor will be stored in `*this` afterwards. `other.rdbuf() -> pbase() == other.rdbuf() -> pptr()` and `other.get_wrapped() == nullptr` are true.

**Remarks**: This constructor disassociates `other` from its wrapped stream buffer, ensuring destruction of `other` produces no output.

```cpp
~basic_syncbuf();
```

**Effects**: Calls `emit()`.
8. **Throws**: Nothing. If an exception is thrown from emit(), the destructor catches and ignores that exception.

### 29.10.2.3 Assignment and swap

```cpp
basic_syncbuf& operator=(basic_syncbuf&& rhs) noexcept;
```

1. **Effects**: Calls emit() then move assigns from rhs. After the move assignment *this has the observable state it would have had if it had been move constructed from rhs (29.10.2.2).

2. **Postconditions**:

   - rhs.get_wrapped() == nullptr is true.
   - this->get_allocator() == rhs.get_allocator() is true when allocator_traits<Allocator>::propagate_on_container_move_assignment::value is true; otherwise, the allocator is unchanged.

3. **Returns**: *this.

4. **Remarks**: This assignment operator disassociates rhs from its wrapped stream buffer, ensuring destruction of rhs produces no output.

```cpp
void swap(basic_syncbuf& other) noexcept;
```

5. ** Preconditions**: Either allocator_traits<Allocator>::propagate_on_container_swap::value is true or this->get_allocator() == other.get_allocator() is true.

6. **Effects**: Exchanges the state of *this and other.

### 29.10.2.4 Member functions

```cpp
bool emit();
```

1. **Effects**: Atomically transfers the associated output of *this to the stream buffer *wrapped, so that it appears in the output stream as a contiguous sequence of characters. wrapped->pubsync() is called if and only if a call was made to sync() since the most recent call to emit(), if any.

2. **Postconditions**: On success, the associated output is empty.

3. **Returns**: true if all of the following conditions hold; otherwise false:

   - wrapped == nullptr is false.
   - All of the characters in the associated output were successfully transferred.
   - The call to wrapped->pubsync() (if any) succeeded.

4. **Synchronization**: All emit() calls transferring characters to the same stream buffer object appear to execute in a total order consistent with the “happens before” relation (6.9.2.2), where each emit() call synchronizes with subsequent emit() calls in that total order.

5. **Remarks**: May call member functions of wrapped while holding a lock uniquely associated with wrapped.

```cpp
streambuf_type* get_wrapped() const noexcept;
```

6. **Returns**: wrapped.

```cpp
allocator_type get_allocator() const noexcept;
```

7. **Returns**: A copy of the allocator that was set in the constructor or assignment operator.

```cpp
void set.emit_on_sync(bool b) noexcept;
```

8. **Effects**: emit_on_sync = b.

### 29.10.2.5 Overridden virtual functions

```cpp
int sync() override;
```

1. **Effects**: Records that the wrapped stream buffer is to be flushed. Then, if emit_on_sync is true, calls emit().

   [Note 1: If emit_on_sync is false, the actual flush is delayed until a call to emit(). — end note]

2. **Returns**: If emit() was called and returned false, returns -1; otherwise 0.
29.10.2.6 Specialized algorithms

template<class charT, class traits, class Allocator>
void swap(basic_syncbuf<charT, traits, Allocator>& a,
        basic_syncbuf<charT, traits, Allocator>& b) noexcept;

1 Effects: Equivalent to a.swap(b).

29.10.3 Class template basic_osyncstream

29.10.3.1 Overview

namespace std {
    template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
    class basic_osyncstream : public basic_ostream<charT, traits> {
        public:
            using char_type = charT;
            using int_type = typename traits::int_type;
            using pos_type = typename traits::pos_type;
            using off_type = typename traits::off_type;
            using traits_type = traits;
            using allocator_type = Allocator;
            using streambuf_type = basic_streambuf<charT, traits>;
            using syncbuf_type = basic_syncbuf<charT, traits, Allocator>;

            basic_osyncstream(streambuf_type*, const Allocator&);
            explicit basic_osyncstream(streambuf_type* obuf)
                : basic_osyncstream(obuf, Allocator()) {}
            basic_osyncstream(basic_ostream<charT, traits>& os, const Allocator& allocator)
                : basic_osyncstream(os.rdbuf(), allocator) {}
            explicit basic_osyncstream(basic_ostream<charT, traits>&& os)
                : basic_osyncstream(os, Allocator()) {}
            basic_osyncstream(basic_osyncstream&&) noexcept;
            basic_osyncstream();

            // 29.10.3.2, construction and destruction
            streambuf_type* get_wrapped() const noexcept;
            syncbuf_type* rdbuf() const noexcept { return const_cast<syncbuf_type*>(addressof(sb)); }

            private:
                syncbuf_type sb; // exposition only
    };
}

1 Allocator shall meet the Cpp17Allocator requirements (Table 36).

2 [Example 1: A named variable can be used within a block statement for streaming.]

    { osyncstream bout(cout);
      bout << "Hello, ";
      bout << "World!";
      bout << endl; // flush is noted
      bout << "and more!\n";
    } // characters are transferred and cout is flushed
    — end example

3 [Example 2: A temporary object can be used for streaming within a single statement.]

    osyncstream(cout) << "Hello, " << "World!" << '\n';

In this example, cout is not flushed. — end example

§ 29.10.3.1
29.10.3.2 Construction and destruction

basic_osyncstream(streambuf_type* buf, const Allocator& allocator);

*Effects:* Initializes `sb` from `buf` and `allocator`. Initializes the base class with `basic_ostream<charT, traits>(addressof(sb))`.

*Note 1:* The member functions of the provided stream buffer can be called from `emit()` while a lock is held, which might result in a deadlock if used incautiously. —*end note*

*Postconditions:* `get_wrapped() == buf` is `true`.

basic_osyncstream(basic_osyncstream&& other) noexcept;

*Effects:* Move constructs the base class and `sb` from the corresponding subobjects of `other`, and calls `basic_ostream<charT, traits>::set_rdbuf(addressof(sb))`.

*Postconditions:* The value returned by `get_wrapped()` is the value returned by `os.get_wrapped()` prior to calling this constructor. `nullptr == other.get_wrapped()` is `true`.

29.10.3.3 Member functions

void emit();

*Effects:* Calls `sb.emit()`. If that call returns `false`, calls `setstate(ios_base::badbit)`.

*Example 1:* A flush on a `basic_osyncstream` does not flush immediately:

```cpp
{ osyncstream bout(cout);
  bout << "Hello," << '\n'; // no flush
  bout.emit(); // characters transferred; cout not flushed
  bout << "World!" << endl; // flush noted; cout not flushed
  bout.emit(); // characters transferred; cout flushed
  bout << "Greetings." << '\n'; // no flush
  bout.emit(); // characters transferred; cout not flushed
} // characters transferred; cout not flushed
```

—*end example*

*Example 2:* The function `emit()` can be used to handle exceptions from operations on the underlying stream.

```cpp
{ osyncstream bout(cout);
  bout << "Hello, " << "World!" << '\n';
  try {
    bout.emit();
  } catch (...) {
    // handle exception
  }
}
```

—*end example*

streambuf_type* get_wrapped() const noexcept;

*Returns:* `sb.get_wrapped()`.

*Example 3:* Obtaining the wrapped stream buffer with `get_wrapped()` allows wrapping it again with an `osyncstream`. For example,

```cpp
{ osyncstream bout1(cout);
  bout1 << "Hello, ";
  { osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '\n';
  }
  bout1 << "World!" << '\n';
}
```

produces the uninterleaved output

```
Goodbye, Planet!
Hello, World!
```

—*end example*
29.11 File systems

29.11.1 General

Subclause 29.11 describes operations on file systems and their components, such as paths, regular files, and directories.

1 A file system is a collection of files and their attributes.

2 A file is an object within a file system that holds user or system data. Files can be written to, or read from, or both. A file has certain attributes, including type. File types include regular files and directories. Other types of files, such as symbolic links, may be supported by the implementation.

3 A directory is a file within a file system that acts as a container of directory entries that contain information about other files, possibly including other directory files. The parent directory of a directory is the directory that both contains a directory entry for the given directory and is represented by the dot-dot filename (29.11.6.2) in the given directory. The parent directory of other types of files is a directory containing a directory entry for the file under discussion.

4 A link is an object that associates a filename with a file. Several links can associate names with the same file. A hard link is a link to an existing file. Some file systems support multiple hard links to a file. If the last hard link to a file is removed, the file itself is removed.

[Note 1: A hard link can be thought of as a shared-ownership smart pointer to a file. — end note]

A symbolic link is a type of file with the property that when the file is encountered during pathname resolution (29.11.6), a string stored by the file is used to modify the pathname resolution.

[Note 2: Symbolic links are often called symlinks. A symbolic link can be thought of as a raw pointer to a file. If the file pointed to does not exist, the symbolic link is said to be a “dangling” symbolic link. — end note]

29.11.2 Conformance

29.11.2.1 General

1 Conformance is specified in terms of behavior. Ideal behavior is not always implementable, so the conformance subclauses take that into account.

29.11.2.2 POSIX conformance

1 Some behavior is specified by reference to POSIX. How such behavior is actually implemented is unspecified.

[Note 1: This constitutes an “as if” rule allowing implementations to call native operating system or other APIs. — end note]

2 Implementations should provide such behavior as it is defined by POSIX. Implementations shall document any behavior that differs from the behavior defined by POSIX. Implementations that do not support exact POSIX behavior should provide behavior as close to POSIX behavior as is reasonable given the limitations of actual operating systems and file systems. If an implementation cannot provide any reasonable behavior, the implementation shall report an error as specified in 29.11.5.

[Note 2: This allows users to rely on an exception being thrown or an error code being set when an implementation cannot provide any reasonable behavior. — end note]

3 Implementations are not required to provide behavior that is not supported by a particular file system.

[Example 1: The FAT file system used by some memory cards, camera memory, and floppy disks does not support hard links, symlinks, and many other features of more capable file systems, so implementations are not required to support those features on the FAT file system but instead are required to report an error as described above. — end example]

29.11.2.3 Operating system dependent behavior conformance

1 Behavior that is specified as being operating system dependent is dependent upon the behavior and characteristics of an operating system. The operating system an implementation is dependent upon is implementation-defined.

2 It is permissible for an implementation to be dependent upon an operating system emulator rather than the actual underlying operating system.
29.11.2.4 File system race behavior

A file system race is the condition that occurs when multiple threads, processes, or computers interleave access and modification of the same object within a file system. Behavior is undefined if calls to functions provided by subclause 29.11 introduce a file system race.

If the possibility of a file system race would make it unreliable for a program to test for a precondition before calling a function described herein, Preconditions: is not specified for the function.

[Note 1: As a design practice, preconditions are not specified when it is unreasonable for a program to detect them prior to calling the function. — end note]

29.11.3 Requirements

29.11.3.1 General

Throughout subclause 29.11, char, wchar_t, char8_t, char16_t, and char32_t are collectively called encoded character types.

Functions with template parameters named EcharT shall not participate in overload resolution unless EcharT is one of the encoded character types.

Template parameters named InputIterator shall meet the Cpp17InputIterator requirements (23.3.5.3) and shall have a value type that is one of the encoded character types.

[Note 1: Use of an encoded character type implies an associated character set and encoding. Since signed char and unsigned char have no implied character set and encoding, they are not included as permitted types. — end note]

Template parameters named Allocator shall meet the Cpp17Allocator requirements (Table 36).

29.11.3.2 Namespaces and headers

Unless otherwise specified, references to entities described in subclause 29.11 are assumed to be qualified with ::std::filesystem::.

29.11.4 Header <filesystem> synopsis

```cpp
#include <filesystem>  // see 17.11.1

namespace std::filesystem {
    // 29.11.6, paths
class path;

    // 29.11.6.8, path non-member functions
    void swap(path& lhs, path& rhs) noexcept;
    size_t hash_value(const path& p) noexcept;

    // 29.11.7, filesystem errors
class filesystem_error;

    // 29.11.10, directory entries
class directory_entry;

    // 29.11.11, directory iterators
class directory_iterator;

    // 29.11.11.3, range access for directory iterators
directory_iterator begin(directory_iterator iter) noexcept;
directory_iterator end(const directory_iterator&) noexcept;

    // 29.11.12, recursive directory iterators
class recursive_directory_iterator;

    // 29.11.12.3, range access for recursive directory iterators
    recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;
    recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;

    // 29.11.9, file status
class file_status;
```
struct space_info {
    uintmax_t capacity;
    uintmax_t free;
    uintmax_t available;

    friend bool operator==(const space_info&, const space_info&) = default;
};

// 29.11.8, enumerations
enum class file_type;
enum class perms;
enum class perm_options;
enum class copy_options;
enum class directory_options;

using file_time_type = chrono::time_point<chrono::file_clock>;

// 29.11.13, filesystem operations
path absolute(const path& p);
path absolute(const path& p, error_code& ec);

path canonical(const path& p);
path canonical(const path& p, error_code& ec);

void copy(const path& from, const path& to);
void copy(const path& from, const path& to, error_code& ec);
void copy(const path& from, const path& to, copy_options options);
void copy(const path& from, const path& to, copy_options options,
         error_code& ec);

bool copy_file(const path& from, const path& to);
bool copy_file(const path& from, const path& to, error_code& ec);
bool copy_file(const path& from, const path& to, copy_options options);
bool copy_file(const path& from, const path& to, copy_options options,
               error_code& ec);

void copy_symlink(const path& existing_symlink, const path& new_symlink);
void copy_symlink(const path& existing_symlink, const path& new_symlink,
                  error_code& ec) noexcept;

void create_directories(const path& p);
bool create_directories(const path& p, error_code& ec);

void create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;

bool create_directory(const path& p, const path& attributes);
bool create_directory(const path& p, const path& attributes,
                      error_code& ec) noexcept;

void create_directory_symlink(const path& to, const path& new_symlink);
void create_directory_symlink(const path& to, const path& new_symlink,
                             error_code& ec) noexcept;

void create_hard_link(const path& to, const path& new_hard_link);
void create_hard_link(const path& to, const path& new_hard_link,
                      error_code& ec) noexcept;

void create_symlink(const path& to, const path& new_symlink);
void create_symlink(const path& to, const path& new_symlink,
                    error_code& ec) noexcept;

path current_path();
path current_path(error_code& ec);
void current_path(const path& p);
void current_path(const path& p, error_code& ec) noexcept;

bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2, error_code& ec) noexcept;

bool exists(file_status s) noexcept;
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;

uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;

uintmax_t hard_link_count(const path& p);
uintmax_t hard_link_count(const path& p, error_code& ec) noexcept;

bool is_block_file(file_status s) noexcept;
bool is_block_file(const path& p);
bool is_block_file(const path& p, error_code& ec) noexcept;

bool is_character_file(file_status s) noexcept;
bool is_character_file(const path& p);
bool is_character_file(const path& p, error_code& ec) noexcept;

bool is_directory(file_status s) noexcept;
bool is_directory(const path& p);
bool is_directory(const path& p, error_code& ec) noexcept;

bool is_empty(const path& p);
bool is_empty(const path& p, error_code& ec);

bool is_fifo(file_status s) noexcept;
bool is_fifo(const path& p);
bool is_fifo(const path& p, error_code& ec) noexcept;

bool is_other(file_status s) noexcept;
bool is_other(const path& p);
bool is_other(const path& p, error_code& ec) noexcept;

bool is_regular_file(file_status s) noexcept;
bool is_regular_file(const path& p);
bool is_regular_file(const path& p, error_code& ec) noexcept;

bool is_socket(file_status s) noexcept;
bool is_socket(const path& p);
bool is_socket(const path& p, error_code& ec) noexcept;

bool is_symlink(file_status s) noexcept;
bool is_symlink(const path& p);
bool is_symlink(const path& p, error_code& ec) noexcept;

file_time_type last_write_time(const path& p);
file_time_type last_write_time(const path& p, error_code& ec) noexcept;
void last_write_time(const path& p, file_time_type new_time);
void last_write_time(const path& p, file_time_type new_time, error_code& ec) noexcept;

void permissions(const path& p, perms prms, perm_options opts=perm_options::replace);
void permissions(const path& p, perms prms, error_code& ec) noexcept;
void permissions(const path& p, perms prms, perm_options opts, error_code& ec);

path proximate(const path& p, error_code& ec);
path proximate(const path& p, const path& base = current_path());
path proximate(const path& p, const path& base, error_code& ec);
Implementations should ensure that the resolution and range of `file_time_type` reflect the operating system dependent resolution and range of file time values.

### 29.11.5 Error reporting [fs.err.report]

1. Filesystem library functions often provide two overloads, one that throws an exception to report file system errors, and another that sets an `error_code`. 

   [Note 1: This supports two common use cases:
   
   (1.1) Uses where file system errors are truly exceptional and indicate a serious failure. Throwing an exception is an appropriate response.
   
   (1.2) Uses where file system errors are routine and do not necessarily represent failure. Returning an error code is the most appropriate response. This allows application specific error handling, including simply ignoring the error.  
   
   end note]

2. Functions not having an argument of type `error_code` handle errors as follows, unless otherwise specified:
   
   (2.1) When a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, an exception of type `filesystem_error` shall be thrown. For functions with a single path argument, that argument shall be passed to the `filesystem_error` constructor with a single path argument. For functions with two path arguments, the first of these arguments shall be passed to the `filesystem_error` constructor as the `path1` argument, and the second shall be passed as the `path2` argument. The `filesystem_error` constructor’s `error_code` argument is set as appropriate for the specific operating system dependent error.
   
   (2.2) Failure to allocate storage is reported by throwing an exception as described in 16.4.6.13.
   
   (2.3) Destructors throw nothing.
Functions having an argument of type `error_code&` handle errors as follows, unless otherwise specified:

1. If a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, the `error_code&` argument is set as appropriate for the specific operating system dependent error. Otherwise, `clear()` is called on the `error_code&` argument.

29.11.6 Class `path` [fs.class.path]

29.11.6.1 General [fs.class.path.general]

1. An object of class `path` represents a path and contains a pathname. Such an object is concerned only with the lexical and syntactic aspects of a path. The path does not necessarily exist in external storage, and the pathname is not necessarily valid for the current operating system or for a particular file system.

2. [Note 1: `path` is used to support the differences between the string types used by different operating systems to represent pathnames, and to perform conversions between encodings when necessary. — end note]

3. A `path` is a sequence of elements that identify the location of a file within a filesystem. The elements are the `root-name_opt`, `root-directory_opt`, and an optional sequence of `filenames` (29.11.6.2). The maximum number of elements in the sequence is operating system dependent (29.11.2.3).

4. An `absolute path` is a path that unambiguously identifies the location of a file without reference to an additional starting location. The elements of a path that determine if it is absolute are operating system dependent. A `relative path` is a path that is not absolute, and as such, only unambiguously identifies the location of a file when resolved relative to an implied starting location. The elements of a path that determine if it is relative are operating system dependent.

5. [Note 2: Pathnames “.” and “..” are relative paths. — end note]

6. A `pathname` is a character string that represents the name of a path. Pathnames are formatted according to the generic pathname format grammar (29.11.6.2) or according to an operating system dependent `native pathname format` accepted by the host operating system.

Pathname resolution is the operating system dependent mechanism for resolving a `pathname` to a particular file in a file hierarchy. There may be multiple `pathnames` that resolve to the same file.

[Example 1: POSIX specifies the mechanism in section 4.11, Pathname resolution. — end example]

```cpp
namespace std::filesystem {
    class path {
        using value_type = see below;
        using string_type = basic_string<value_type>;
        static constexpr value_type preferred_separator = see below;

        // 29.11.6.5.1, constructors and destructor
        path() noexcept;
        path(const path& p);
        path(path&& p) noexcept;
        path(string_type&& source, format fmt = auto_format);
        template<class Source>
            path(const Source& source, format fmt = auto_format);
        template<class InputIterator>
            path(InputIterator first, InputIterator last, format fmt = auto_format);
        template<class InputIterator>
            path(InputIterator first, InputIterator last, const locale& loc, format fmt = auto_format);
        ~path();

        // 29.11.6.5.2, assignments
        path& operator=(const path& p);
        path& operator=(path&& p) noexcept;
        path& operator=(string_type&& source);
        path& assign(string_type&& source);
```
template<class Source>
    path& operator=(const Source& source);

template<class Source>
    path& assign(const Source& source);

template<class InputIterator>
    path& assign(InputIterator first, InputIterator last);

// 29.11.6.5.3, appends
path& operator/=(const path& p);

// 29.11.6.5.4, concatenation
path& operator+=((const path& x);
path& operator+=(const string_type& x);
path& operator+=(basic_string_view<value_type> x);
path& operator+=(const value_type* x);
path& operator+=(value_type x);

// 29.11.6.5.5, modifiers
void clear() noexcept;
path& make_preferred();
path& remove_filename();
path& replace_filename(const path& replacement);
path& replace_extension(const path& replacement = path());
void swap(path& rhs) noexcept;

// 29.11.6.8, non-member operators
friend bool operator==(const path& lhs, const path& rhs) noexcept;
friend strong_ordering operator<=>(const path& lhs, const path& rhs) noexcept;

friend path operator/((const path& lhs, const path& rhs);

// 29.11.6.5.6, native format observers
const string_type& native() const noexcept;
const value_type* c_str() const noexcept;
operator string_type() const;

// 29.11.6.5.7, generic format observers
template<class EcharT, class traits = char_traits<EcharT>,
    class Allocator = allocator<EcharT>>
    basic_string<EcharT, traits, Allocator>
        string(const Allocator& a = Allocator()) const;
    std::string string() const;
    std::wstring wstring() const;
    std::u8string u8string() const;
    std::u16string u16string() const;
    std::u32string u32string() const;


§ 29.11.6.1
```cpp
generic_string(const Allocator& a = Allocator()) const;
std::string generic_string() const;
std::wstring generic_wstring() const;
std::u8string generic_u8string() const;
std::u16string generic_u16string() const;
std::u32string generic_u32string() const;

// 29.11.6.5.8, compare
int compare(const path& p) const noexcept;
int compare(const string_type& s) const;
int compare(basic_string_view<value_type> s) const;
int compare(const value_type* s) const;

// 29.11.6.5.9, decomposition
path root_name() const;
path root_directory() const;
path relative_path() const;
path parent_path() const;
path filename() const;
path stem() const;
path extension() const;

// 29.11.6.5.10, query
[[nodiscard]] bool empty() const noexcept;
bool has_root_name() const;
bool has_root_directory() const;
bool has_root_path() const;
bool has_relative_path() const;
bool has_parent_path() const;
bool has_filename() const;
bool has_stem() const;
bool has_extension() const;
bool is_absolute() const;
bool is_relative() const;

// 29.11.6.5.11, generation
path lexically_normal() const;
path lexically_relative(const path& base) const;
path lexically_proximate(const path& base) const;

// 29.11.6.6, iterators
class iterator;
using const_iterator = iterator;
iterator begin() const;
iterator end() const;

// 29.11.6.7, path inserter and extractor
template<class charT, class traits>
friend basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const path& p);
template<class charT, class traits>
friend basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, path& p);
```

7 value_type is a typedef for the operating system dependent encoded character type used to represent pathnames.

8 The value of the preferred_separator member is the operating system dependent preferred-separator character (29.11.6.2).
Example 2: For POSIX-based operating systems, value_type is char and preferred_separator is the slash character (’/’). For Windows-based operating systems, value_type is wchar_t and preferred_separator is the backslash character (L’\’). — end example

29.11.6.2 Generic pathname format [fs.path.generic]

pathname:
  root-name_opt root-directory_opt relative-path

root-name:
  operating system dependent sequences of characters
  implementation-defined sequences of characters

root-directory:
  directory-separator

relative-path:
  filename
  directory-separator relative-path
  an empty path

filename:
  non-empty sequence of characters other than directory-separator characters

directory-separator:
  preferred-separator directory-separator
  fallback-separator directory-separator

preferred-separator:
  operating system dependent directory separator character

fallback-separator:
  L’/’

1 A filename is the name of a file. The dot and dot-dot filenames, consisting solely of one and two period characters respectively, have special meaning. The following characteristics of filenames are operating system dependent:

(1.1) — The permitted characters.

[Example 1: Some operating systems prohibit the ASCII control characters (0x00 – 0x1F) in filenames. — end example]

[Note 1: Wider portability can be achieved by limiting filename characters to the POSIX Portable Filename Character Set:
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  0 1 2 3 4 5 6 7 8 9 . _ - — end note]

(1.2) — The maximum permitted length.

(1.3) — Filenames that are not permitted.

(1.4) — Filenames that have special meaning.

(1.5) — Case awareness and sensitivity during path resolution.

(1.6) — Special rules that may apply to file types other than regular files, such as directories.

2 Except in a root-name, multiple successive directory-separator characters are considered to be the same as one directory-separator character.

3 The dot filename is treated as a reference to the current directory. The dot-dot filename is treated as a reference to the parent directory. What the dot-dot filename refers to relative to root-directory is implementation-defined. Specific filenames may have special meanings for a particular operating system.

4 A root-name identifies the starting location for pathname resolution (29.11.6). If there are no operating system dependent root-names, at least one implementation-defined root-name is required.

[Note 2: Many operating systems define a name beginning with two directory-separator characters as a root-name that identifies network or other resource locations. Some operating systems define a single letter followed by a colon as a drive specifier – a root-name identifying a specific device such as a disk drive. — end note]

5 If a root-name is otherwise ambiguous, the possibility with the longest sequence of characters is chosen.

[Note 3: On a POSIX-like operating system, it is impossible to have a root-name and a relative-path without an intervening root-directory element. — end note]

§ 29.11.6.2 1466
Normalization of a generic format pathname means:

1. If the path is empty, stop.
2. Replace each slash character in the root-name with a preferred-separator.
3. Replace each directory-separator with a preferred-separator.
   [Note 4: The generic pathname grammar defines directory-separator as one or more slashes and preferred-separators. — end note]
4. Remove each dot filename and any immediately following directory-separator.
5. As long as any appear, remove a non-dot-dot filename immediately followed by a directory-separator and a dot-dot filename, along with any immediately following directory-separator.
6. If there is a root-directory, remove all dot-dot filenames and any directory-separators immediately following them.
   [Note 5: These dot-dot filenames attempt to refer to nonexistent parent directories. — end note]
7. If the last filename is dot-dot, remove any trailing directory-separator.
8. If the path is empty, add a dot.

The result of normalization is a path in normal form, which is said to be normalized.

29.11.6.3 Conversions

29.11.6.3.1 Argument format conversions

Several functions are defined to accept detected-format arguments, which are character sequences. A detected-format argument represents a path using either a pathname in the generic format (29.11.6.2) or a pathname in the native format (29.11.6). Such an argument is taken to be in the generic format if and only if it matches the generic format and is not acceptable to the operating system as a native path.

[Note 2: Some operating systems have no unambiguous way to distinguish between native format and generic format arguments. This is by design as it simplifies use for operating systems that do not require disambiguation. An implementation for an operating system where disambiguation is required is permitted to distinguish between the formats. — end note]

Pathnames are converted as needed between the generic and native formats in an operating-system-dependent manner. Let \( G(n) \) and \( N(g) \) in a mathematical sense be the implementation’s functions that convert native-to-generic and generic-to-native formats respectively. If \( g=G(n) \) for some \( n \), then \( G(N(g))=g \); if \( n=N(g) \) for some \( g \), then \( N(G(n))=n \).

[Note 3: Neither \( G \) nor \( N \) need be invertible. — end note]

If the native format requires paths for regular files to be formatted differently from paths for directories, the path shall be treated as a directory path if its last element is a directory-separator, otherwise it shall be treated as a path to a regular file.

[Note 4: A path stores a native format pathname (29.11.6.5.6) and acts as if it also stores a generic format pathname, related as given below. The implementation can generate the generic format pathname based on the native format pathname (and possibly other information) when requested. — end note]

When a path is constructed from or is assigned a single representation separate from any path, the other representation is selected by the appropriate conversion function (\( G \) or \( N \)).

When the (new) value \( p \) of one representation of a path is derived from the representation of that or another path, a value \( q \) is chosen for the other representation. The value \( q \) converts to \( p \) (by \( G \) or \( N \) as appropriate) if any such value does so; \( q \) is otherwise unspecified.

[Note 5: If \( q \) is the result of converting any path at all, it is the result of converting \( p \). — end note]
29.11.6.3.2 Type and encoding conversions

The native encoding of an ordinary character string is the operating system dependent current encoding for pathnames (29.11.6). The native encoding for wide character strings is the implementation-defined execution wide-character set encoding (5.3).

For member function arguments that take character sequences representing paths and for member functions returning strings, value type and encoding conversion is performed if the value type of the argument or return value differs from path::value_type. For the argument or return value, the method of conversion and the encoding to be converted to is determined by its value type:

(2.1) — char: The encoding is the native ordinary encoding. The method of conversion, if any, is operating system dependent.

[Note 1: For POSIX-based operating systems path::value_type is char so no conversion from char value type arguments or to char value type return values is performed. For Windows-based operating systems, the native ordinary encoding is determined by calling a Windows API function. — end note]

[Note 2: This results in behavior identical to other C and C++ standard library functions that perform file operations using ordinary character strings to identify paths. Changing this behavior would be surprising and error prone. — end note]

(2.2) — wchar_t: The encoding is the native wide encoding. The method of conversion is unspecified.

[Note 3: For Windows-based operating systems path::value_type is wchar_t so no conversion from wchar_t value type arguments or to wchar_t value type return values is performed. — end note]

(2.3) — char8_t: The encoding is UTF-8. The method of conversion is unspecified.

(2.4) — char16_t: The encoding is UTF-16. The method of conversion is unspecified.

(2.5) — char32_t: The encoding is UTF-32. The method of conversion is unspecified.

If the encoding being converted to has no representation for source characters, the resulting converted characters, if any, are unspecified. Implementations should not modify member function arguments if already of type path::value_type.

29.11.6.4 Requirements

In addition to the requirements (29.11.3), function template parameters named Source shall be one of:

(1.1) — basic_string<EcharT, traits, Allocator>. A function argument const Source& source shall have an effective range [source.begin(), source.end()).

(1.2) — basic_string_view<EcharT, traits>. A function argument const Source& source shall have an effective range [source.begin(), source.end()).

(1.3) — A type meeting the Cpp17InputIterator requirements that iterates over a NTCTS. The value type shall be an encoded character type. A function argument const Source& source shall have an effective range [source, end) where end is the first iterator value with an element value equal to iterator_traits<Source>::value_type().

(1.4) — A character array that after array-to-pointer decay results in a pointer to the start of a NTCTS. The value type shall be an encoded character type. A function argument const Source& source shall have an effective range [source, end) where end is the first iterator value with an element value equal to iterator_traits<decay_t<Source>>::value_type().

Functions taking template parameters named Source shall not participate in overload resolution unless Source denotes a type other than path, and either

(2.1) — Source is a specialization of basic_string or basic_string_view, or

(2.2) — the qualified-id iterator_traits<decay_t<Source>>::value_type is valid and denotes a possibly const encoded character type (13.10.3).

[Note 1: See path conversions (29.11.6.3) for how the value types above and their encodings convert to path::value_type and its encoding. — end note]

Arguments of type Source shall not be null pointers.
29.11.6.5 Members

29.11.6.5.1 Constructors

path() noexcept;

Postconditions: empty() == true.

path(const path& p);
path(path&& p) noexcept;

Effects: Constructs an object of class path having the same pathname in the native and generic formats, respectively, as the original value of p. In the second form, p is left in a valid but unspecified state.

path(string_type&& source, format fmt = auto_format);

Effects: Constructs an object of class path for which the pathname in the detected-format of source has the original value of source (29.11.6.3.1), converting format if required (29.11.6.3.1). source is left in a valid but unspecified state.

template<class Source>
path(const Source& source, format fmt = auto_format);
template<class InputIterator>
path(InputIterator first, InputIterator last, format fmt = auto_format);

Effects: Let s be the effective range of source (29.11.6.4) or the range [first, last), with the encoding converted if required (29.11.6.3). Finds the detected-format of s (29.11.6.3.1) and constructs an object of class path for which the pathname in that format is s.

template<class Source>
path(const Source& source, const locale& loc, format fmt = auto_format);
template<class InputIterator>
path(InputIterator first, InputIterator last, const locale& loc, format fmt = auto_format);

Mandates: The value type of Source and InputIterator is char.

Effects: Let s be the effective range of source or the range [first, last), after converting the encoding as follows:

(6.1) If value_type is wchar_t, converts to the native wide encoding (29.11.6.3.2) using the codecvt<wchar_t, char, mbstate_t> facet of loc.

(6.2) Otherwise a conversion is performed using the codecvt<wchar_t, char, mbstate_t> facet of loc, and then a second conversion to the current ordinary encoding.

Finds the detected-format of s (29.11.6.3.1) and constructs an object of class path for which the pathname in that format is s.

[Example 1: A string is to be read from a database that is encoded in ISO/IEC 8859-1, and used to create a directory:

```cpp
namespace fs = std::filesystem;
std::string latin1_string = read_latin1_data();
codecvt_8859_1<wchar_t> latin1_facet;
std::locale latin1_locale(std::locale(), latin1_facet);
fs::create_directory(fs::path(latin1_string, latin1_locale));
```

For POSIX-based operating systems, the path is constructed by first using latin1_facet to convert ISO/IEC 8859-1 encoded latin1_string to a wide character string in the native wide encoding (29.11.6.3.2). The resulting wide string is then converted to an ordinary character pathname string in the current native ordinary encoding. If the native wide encoding is UTF-16 or UTF-32, and the current native ordinary encoding is UTF-8, all of the characters in the ISO/IEC 8859-1 character set will be converted to their Unicode representation, but for other native ordinary encodings some characters may have no representation.

For Windows-based operating systems, the path is constructed by using latin1_facet to convert ISO/IEC 8859-1 encoded latin1_string to a UTF-16 encoded wide character pathname string. All of the characters in the ISO/IEC 8859-1 character set will be converted to their Unicode representation. — end example]
29.11.6.5.2 Assignments

```cpp
path& operator=(const path& p);  

Effects: If *this and p are the same object, has no effect. Otherwise, sets both respective pathnames of *this to the respective pathnames of p.

Returns: *this.
```

```cpp
path& operator=(path&& p) noexcept;
```

```cpp
Effects: If *this and p are the same object, has no effect. Otherwise, sets both respective pathnames of *this to the respective pathnames of p. p is left in a valid but unspecified state.

[Note 1: A valid implementation is swap(p). — end note]

Returns: *this.
```

```cpp
path& operator=(string_type&& source);
```

```cpp
Effects: Sets the pathname in the detected-format of source to the original value of source. source is left in a valid but unspecified state.

Returns: *this.
```

```cpp
template<class Source>
path& operator=(const Source& source);
```

```cpp
template<class Source>
path& assign(const Source& source);
```

```cpp
template<class InputIterator>
path& assign(InputIterator first, InputIterator last);
```

```cpp
Effects: Let s be the effective range of source (29.11.6.4) or the range [first, last), with the encoding converted if required (29.11.6.3). Finds the detected-format of s (29.11.6.3.1) and sets the pathname in that format to s.

Returns: *this.
```

29.11.6.5.3 Appends

```cpp
path& operator/=(const path& p);
```

```cpp
Effects: If p.is_absolute() || (p.has_root_name() & p.root_name() != root_name()), then operator=(p).

Otherwise, modifies *this as if by these steps:

- If p.has_root_directory(), then removes any root directory and relative path from the generic format pathname. Otherwise, if has_root_directory() & is_absolute() is true or if has_root_filename() is true, then appends path::preferred_separator to the generic format pathname.

- Then appends the native format pathname of p, omitting any root-name from its generic format pathname, to the native format pathname.

[Example 1: Even if //host is interpreted as a root-name, both of the paths path("//host")/"foo" and path("//host")/"foo" equal "/host/foo" (although the former might use backslash as the preferred separator).

Expression examples:

// On POSIX,
path("foo") /= path("""); // yields path("foo")
path("foo") /= path("/bar"); // yields path("/bar")

// On Windows,
path("foo") /= path("""); // yields path("foo")
path("foo") /= path("/bar"); // yields path("/bar")
path("foo") /= path("c:/bar"); // yields path("c:/bar")
path("foo") /= path("c:"); // yields path("c:")
```
path("c:" /= path(""));  // yields path("c:")
path("c:foo") /= path("/bar");  // yields path("c:/bar")
path("c:foo") /= path("c:bar");  // yields path("c:foo\bar")

—end example

Returns: *this.

```cpp
template<class Source>
path& operator/(const Source& source);
template<class Source>
path& append(const Source& source);
```

Effects: Equivalent to: return operator/(path(source));

```cpp
template<class InputIterator>
path& append(InputIterator first, InputIterator last);
```

Effects: Equivalent to: return operator/(path(first, last));

29.11.6.5.4 Concatenation

```cpp
path& operator+=(const path& x);
path& operator+=(const string_type& x);
path& operator+=(basic_string_view<value_type> x);
path& operator+=(const value_type* x);
template<class Source>
path& operator+=(const Source& x);
template<class Source>
path& concat(const Source& x);
```

Effects: Appends path(x).native() to the pathname in the native format.

[Note 1: This directly manipulates the value of native() and might not be portable between operating systems.
—end note]

Returns: *this.

```cpp
path& operator+=(value_type x);
template<class EcharT>
path& operator+=(EcharT x);
```

Effects: Equivalent to: return *this += basic_string_view(&x, 1);

```cpp
template<class InputIterator>
path& concat(InputIterator first, InputIterator last);
```

Effects: Equivalent to: return *this += path(first, last);

29.11.6.5.5 Modifiers

```cpp
void clear() noexcept;
```

Postconditions: empty() == true.

```cpp
path& make_preferred();
```

Effects: Each directory-separator of the pathname in the generic format is converted to preferred-separator.

Returns: *this.

[Example 1:

```cpp
path p("foo/bar");
std::cout << p << '\n';
p.make_preferred();
std::cout << p << '\n';
```

On an operating system where preferred-separator is a slash, the output is:

"foo/bar"

On an operating system where preferred-separator is a backslash, the output is:...]
```cpp
path& remove_filename();

Effects: Remove the generic format pathname of filename() from the generic format pathname.
Postconditions: !has_filename().
Returns: *this.

[Example 2:
path("foo/bar").remove_filename(); // yields "foo/
path("foo/").remove_filename();    // yields "foo/
path("/foo").remove_filename();    // yields "/"
path("/").remove_filename();       // yields "/
— end example]

path& replace_filename(const path& replacement);

Effects: Equivalent to:
remove_filename();
operator/=(replacement);
Returns: *this.

[Example 3:
path("/foo").replace_filename("bar"); // yields "/bar" on POSIX
path("/").replace_filename("bar");     // yields "/bar" on POSIX
— end example]

path& replace_extension(const path& replacement = path());

Effects:
(12.1) — Any existing extension() (29.11.5.9) is removed from the pathname in the generic format, then
(12.2) — If replacement is not empty and does not begin with a dot character, a dot character is appended
to the pathname in the generic format, then
(12.3) — operator+=(replacement);
Returns: *this.

void swap(path& rhs) noexcept;
Effects: Swaps the contents (in all formats) of the two paths.
Complexity: Constant time.

29.11.6.5.6 Native format observers

The string returned by all native format observers is in the native pathname format (29.11.6).

const string_type& native() const noexcept;
Returns: The pathname in the native format.

const value_type* c_str() const noexcept;
Effects: Equivalent to: return native().c_str();
operator string_type() const;
Returns: native().

[Note 1: Conversion to string_type is provided so that an object of class path can be given as an argument to
existing standard library file stream constructors and open functions. — end note]

template<class EcharT, class traits = char_traits<EcharT>,
    class Allocator = allocator<EcharT>>
basic_string<EcharT, traits, Allocator>

§ 29.11.6.5.6 1472
```
string(const Allocator& a = Allocator()) const;

Returns: native().

Remarks: All memory allocation, including for the return value, shall be performed by a. Conversion, if any, is specified by 29.11.6.3.

std::string string() const;
std::wstring wstring() const;
std::u8string u8string() const;
std::u16string u16string() const;
std::u32string u32string() const;

Returns: native().

Remarks: Conversion, if any, is performed as specified by 29.11.6.3.

29.11.6.5.7 Generic format observers

Generic format observer functions return strings formatted according to the generic pathname format (29.11.6.2). A single slash ('/') character is used as the directory-separator.

[Example 1: On an operating system that uses backslash as its preferred-separator,

path("foo\bar").generic_string()

returns "foo/bar". — end example]

template<class EcharT, class traits = char_traits<EcharT>,
class Allocator = allocator<EcharT>>
basic_string<EcharT, traits, Allocator>
generic_string(const Allocator& a = Allocator()) const;

Returns: The pathname in the generic format.

Remarks: All memory allocation, including for the return value, shall be performed by a. Conversion, if any, is specified by 29.11.6.3.

std::string generic_string() const;
std::wstring generic_wstring() const;
std::u8string generic_u8string() const;
std::u16string generic_u16string() const;
std::u32string generic_u32string() const;

Returns: The pathname in the generic format.

Remarks: Conversion, if any, is specified by 29.11.6.3.

29.11.6.5.8 Compare

int compare(const path& p) const noexcept;

Returns:

(1.1) — Let rootNameComparison be the result of this->root_name().native().compare(p.root_name().native()). If rootNameComparison is not 0, rootNameComparison.

(1.2) — Otherwise, if !this->has_root_directory() and p.has_root_directory(), a value less than 0.

(1.3) — Otherwise, if this->has_root_directory() and !p.has_root_directory(), a value greater than 0.

(1.4) — Otherwise, if native() for the elements of this->relative_path() are lexicographically less than native() for the elements of p.relative_path(), a value less than 0.

(1.5) — Otherwise, if native() for the elements of this->relative_path() are lexicographically greater than native() for the elements of p.relative_path(), a value greater than 0.

(1.6) — Otherwise, 0.

int compare(const string_type& s) const;
int compare(basic_string_view<value_type> s) const;

§ 29.11.6.5.8 1473
int compare(const value_type* s) const;

Effects: Equivalent to: return compare(path(s));

29.11.6.5.9 Decomposition

path root_name() const;

Returns: root-name, if the pathname in the generic format includes root-name, otherwise path().

path root_directory() const;

Returns: root-directory, if the pathname in the generic format includes root-directory, otherwise path().

path root_path() const;

Returns: root_name() / root_directory().

path relative_path() const;

Returns: A path composed from the pathname in the generic format, if empty() is false, beginning with the first filename after root_path(). Otherwise, path().

path parent_path() const;

Returns: *this if has_relative_path() is false, otherwise a path whose generic format pathname is the longest prefix of the generic format pathname of *this that produces one fewer element in its iteration.

path filename() const;

Returns: relative_path().empty() ? path() : *--end().

[Example 1:

```
path("/foo/bar.txt").filename(); // yields "bar.txt"
path("/foo/bar").filename(); // yields "bar"
path("/foo/bar/").filename(); // yields ""
path("/").filename(); // yields ""
path("//host").filename(); // yields ""
path(".").filename(); // yields "."
path(".." ).filename(); // yields "."
```
—end example]

path stem() const;

Returns: Let f be the generic format pathname of filename(). Returns a path whose pathname in the generic format is

- f, if it contains no periods other than a leading period or consists solely of one or two periods;
- otherwise, the prefix of f ending before its last period.

[Example 2:

```
std::cout << path("/foo/bar.txt").stem(); // outputs "bar"
path p = "foo.bar.baz.tar";
for (; !p.extension().empty(); p = p.stem())
    std::cout << p.extension() << \\
    //outputs: .tar
    //.
    //.baz
    //.
```
—end example]

path extension() const;

Returns: A path whose pathname in the generic format is the suffix of filename() not included in stem().

[Example 3:

```
path("/foo/bar.txt").extension(); // yields ".txt" and stem() is "bar"
path("/foo/bar").extension(); // yields "" and stem() is "bar"
path("/foo/.profile").extension(); // yields "" and stem() is ".profile"
```
path(".bar").extension(); // yields "" and stem() is ".bar"
path("..bar").extension(); // yields ".bar" and stem() is ".

— end example

[Note 1: The period is included in the return value so that it is possible to distinguish between no extension and an empty extension. — end note]

[Note 2: On non-POSIX operating systems, for a path p, it is possible that p.stem() + p.extension() == p.filename() is false, even though the generic format pathnames are the same. — end note]

29.11.6.5.10 Query \[fs.path.query\]
[[nodiscard]] bool empty() const noexcept;

Returns: true if the pathname in the generic format is empty, otherwise false.

bool has_root_path() const;

Returns: !root_path().empty().

bool has_root_name() const;

Returns: !root_name().empty().

bool has_root_directory() const;

Returns: !root_directory().empty().

bool has_relative_path() const;

Returns: !relative_path().empty().

bool has_parent_path() const;

Returns: !parent_path().empty().

bool has_filename() const;

Returns: !filename().empty().

bool has_stem() const;

Returns: !stem().empty().

bool has_extension() const;

Returns: !extension().empty().

bool is_absolute() const;

Returns: true if the pathname in the native format contains an absolute path (29.11.6), otherwise false.

[Example 1: path("/").is_absolute() is true for POSIX-based operating systems, and false for Windows-based operating systems. — end example]

bool is_relative() const;

Returns: !is_absolute().

29.11.6.5.11 Generation \[fs.path.gen\]

path lexically_normal() const;

Returns: A path whose pathname in the generic format is the normal form (29.11.6.2) of the pathname in the generic format of *this.

[Example 1:

assert(path("foo=./bar/..").lexically_normal() == "foo/");
assert(path("foo=./bar/../").lexically_normal() == "foo/");

The above assertions will succeed. On Windows, the returned path's directory-separator characters will be backslashes rather than slashes, but that does not affect path equality. — end example]
path lexically_relative(const path& base) const;

Effects: If:

(3.1) root_name() != base.root_name() is true, or
(3.2) is_absolute() != base.is_absolute() is true, or
(3.3) !has_root_directory() && base.has_root_directory() is true, or
(3.4) any filename in relative_path() or base.relative_path() can be interpreted as a root-name,

returns path().

[Note 1: On a POSIX implementation, no filename in a relative-path is acceptable as a root-name. — end note]

Determines the first mismatched element of *this and base as if by:

auto [a, b] = mismatch(begin(), end(), base.begin(), base.end());

Then,

(3.5) if a == end() and b == base.end(), returns path("."); otherwise
(3.6) let n be the number of filename elements in [b, base.end()) that are not dot or dot-dot or empty, minus the number that are dot-dot. If n<0, returns path(); otherwise
(3.7) if n == 0 and (a == end() || a->empty()), returns path("."); otherwise
(3.8) returns an object of class path that is default-constructed, followed by
(3.8.1) application of operator/(path(".")) n times, and then
(3.8.2) application of operator/= for each element in [a, end()).

4 Returns: *this made relative to base. Does not resolve (29.11.6) symlinks. Does not first normalize (29.11.6.2) *this or base.

[Example 2:

assert(path("/a/d").lexically_relative("/a/b/c") == "../../d");
assert(path("/a/b/c").lexically_relative("/a/d") == ".../b/c");
assert(path("a/b/c").lexically_relative("a") == "b/c");
assert(path("a/b/c").lexically_relative("a/b/c/x/y") == "..../.");
assert(path("a/b/c").lexically_relative("a/b/c") == ".");
assert(path("a/b").lexically_relative("c/d") == ".../a/b");

The above assertions will succeed. On Windows, the returned path’s directory-separator characters will be backslashes rather than slashes, but that does not affect path equality. — end example]

[Note 2: If symlink following semantics are desired, use the operational function relative(). — end note]

[Note 3: If normalization (29.11.6.2) is needed to ensure consistent matching of elements, apply lexically_normal() to *this, base, or both. — end note]

path lexically_proximate(const path& base) const;

Returns: If the value of lexically_relative(base) is not an empty path, return it. Otherwise return *this.

[Note 4: If symlink following semantics are desired, use the operational function proximate(). — end note]

[Note 5: If normalization (29.11.6.2) is needed to ensure consistent matching of elements, apply lexically_normal() to *this, base, or both. — end note]

29.11.6.6 Iterators [fs.path.itr]

Path iterators iterate over the elements of the pathname in the generic format (29.11.6.2).

A path::iterator is a constant iterator meeting all the requirements of a bidirectional iterator (23.3.5.6) except that, for dereferenceable iterators a and b of type path::iterator with a == b, there is no requirement that *a and *b are bound to the same object. Its value_type is path.

Calling any non-const member function of a path object invalidates all iterators referring to elements of that object.

For the elements of the pathname in the generic format, the forward traversal order is as follows:

(4.1) — The root-name element, if present.
The root-directory element, if present.

[Note 1: The generic format is required to ensure lexicographical comparison works correctly. — end note]

Each successive filename element, if present.

An empty element, if a trailing non-root directory-separator is present.

The backward traversal order is the reverse of forward traversal.

iterator begin() const;

Returns: An iterator for the first present element in the traversal list above. If no elements are present, the end iterator.

iterator end() const;

Returns: The end iterator.

29.11.6.7 Inserter and extractor

[fs.path.io]

```cpp
template<class charT, class traits>
friend basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const path& p);
```

Effects: Equivalent to os << quoted(p.string<charT, traits>())).

[Note 1: The quoted function is described in 29.7.8. — end note]

Returns: os.

```cpp
template<class charT, class traits>
friend basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, path& p);
```

Effects: Equivalent to:
```
basic_string<charT, traits> tmp;
is >> quoted(tmp);
p = tmp;
```

Returns: is.

29.11.6.8 Non-member functions

[fs.path.nonmember]

```cpp
void swap(path& lhs, path& rhs) noexcept;
```

Effects: Equivalent to lhs.swap(rhs).

```cpp
size_t hash_value(const path& p) noexcept;
```

Returns: A hash value for the path p. If for two paths, p1 == p2 then hash_value(p1) == hash_value(p2).

```cpp
friend bool operator==(const path& lhs, const path& rhs) noexcept;
```

Returns: lhs.compare(rhs) == 0.

[Note 1: Path equality and path equivalence have different semantics.

Equality is determined by the path non-member operator==, which considers the two paths’ lexical representations only.

[Example 1: path("foo") == "bar" is never true. — end example]

Equivalence is determined by the equivalent() non-member function, which determines if two paths resolve (29.11.6) to the same file system entity.

[Example 2: equivalent("foo", "bar") will be true when both paths resolve to the same file. — end example]

— end note]

```cpp
friend strong_ordering operator<=>(const path& lhs, const path& rhs) noexcept;
```

Returns: lhs.compare(rhs) <=> 0.

§ 29.11.6.8 1477
friend path operator/(const path& lhs, const path& rhs);

Effects: Equivalent to: return path(lhs) /= rhs;

29.11.7 Class filesystem_error

29.11.7.1 General

namespace std::filesystem {
    class filesystem_error : public system_error {
    public:
        filesystem_error(const string& what_arg, error_code ec);
        filesystem_error(const string& what_arg,
                         const path& p1, error_code ec);
        filesystem_error(const string& what_arg,
                         const path& p1, const path& p2, error_code ec);

        const path& path1() const noexcept;
        const path& path2() const noexcept;
        const char* what() const noexcept override;
    }
}

1 The class filesystem_error defines the type of objects thrown as exceptions to report file system errors from functions described in subclause 29.11.

29.11.7.2 Members

1 Constructors are provided that store zero, one, or two paths associated with an error.

filesystem_error(const string& what_arg, error_code ec);

2 Postconditions:
(2.1) code() == ec,
(2.2) path1().empty() == true,
(2.3) path2().empty() == true, and
(2.4) string_view(what()).find(what_arg.c_str()) != string_view::npos.

filesystem_error(const string& what_arg, const path& p1, error_code ec);

3 Postconditions:
(3.1) code() == ec,
(3.2) path1() returns a reference to the stored copy of p1,
(3.3) path2().empty() == true, and
(3.4) string_view(what()).find(what_arg.c_str()) != string_view::npos.

filesystem_error(const string& what_arg, const path& p1, const path& p2, error_code ec);

4 Postconditions:
(4.1) code() == ec,
(4.2) path1() returns a reference to the stored copy of p1,
(4.3) path2() returns a reference to the stored copy of p2, and
(4.4) string_view(what()).find(what_arg.c_str()) != string_view::npos.

const path& path1() const noexcept;

Returns: A reference to the copy of p1 stored by the constructor, or, if none, an empty path.

const path& path2() const noexcept;

Returns: A reference to the copy of p2 stored by the constructor, or, if none, an empty path.
const char* what() const noexcept override;

Returns: An nstb that incorporates the what_arg argument supplied to the constructor. The exact format is unspecified. Implementations should include the system_error::what() string and the pathnames of path1 and path2 in the native format in the returned string.

29.11.8 Enumerations

29.11.8.1 Enum path::format

This enum specifies constants used to identify the format of the character sequence, with the meanings listed in Table 128.

Table 128: Enum path::format

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>native_format</td>
<td>The native pathname format.</td>
</tr>
<tr>
<td>generic_format</td>
<td>The generic pathname format.</td>
</tr>
<tr>
<td>auto_format</td>
<td>The interpretation of the format of the character sequence is implementation-defined. The implementation may inspect the content of the character sequence to determine the format. Recommended practice: For POSIX-based systems, native and generic formats are equivalent and the character sequence should always be interpreted in the same way.</td>
</tr>
</tbody>
</table>

29.11.8.2 Enum class file_type

This enum class specifies constants used to identify file types, with the meanings listed in Table 129. The values of the constants are distinct.

Table 129: Enum class file_type

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>The type of the file has not been determined or an error occurred while trying to determine the type.</td>
</tr>
</tbody>
</table>
| not_found       | Pseudo-type indicating the file was not found.  
|                 | [Note 1: The file not being found is not considered an error while determining the type of a file. — end note] |
| regular         | Regular file                                                            |
| directory       | Directory file                                                          |
| symlink         | Symbolic link file                                                      |
| block           | Block special file                                                      |
| character       | Character special file                                                  |
| fifo            | FIFO or pipe file                                                       |
| socket          | Socket file                                                             |
| implementation-defined | Implementations that support file systems having file types in addition to the above file_type types shall supply implementation-defined file_type constants to separately identify each of those additional file types |
| unknown         | The file exists but the type could not be determined                    |

29.11.8.3 Enum class copy_options

The enum class type copy_options is a bitmask type (16.3.3.3.4) that specifies bitmask constants used to control the semantics of copy operations. The constants are specified in option groups with the meanings listed in Table 130. The constant none represents the empty bitmask, and is shown in each option group for purposes of exposition; implementations shall provide only a single definition. Every other constant in the table represents a distinct bitmask element.

§ 29.11.8.3
Table 130: Enum class copy_options

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Error; file already exists.</td>
</tr>
<tr>
<td>skip_existing</td>
<td>Do not overwrite existing file, do not report an error.</td>
</tr>
<tr>
<td>overwrite_existing</td>
<td>Overwrite the existing file.</td>
</tr>
<tr>
<td>update_existing</td>
<td>Overwrite the existing file if it is older than the replacement file.</td>
</tr>
</tbody>
</table>

Option group controlling copy function effects for existing target files

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Do not copy sub-directories.</td>
</tr>
<tr>
<td>recursive</td>
<td>Recursively copy sub-directories and their contents.</td>
</tr>
</tbody>
</table>

Option group controlling copy function effects for symbolic links

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Follow symbolic links.</td>
</tr>
<tr>
<td>copy_symlinks</td>
<td>Copy symbolic links as symbolic links rather than copying the files that they point to.</td>
</tr>
<tr>
<td>skip_symlinks</td>
<td>Ignore symbolic links.</td>
</tr>
</tbody>
</table>

Option group controlling copy function effects for choosing the form of copying

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Copy content.</td>
</tr>
<tr>
<td>directories_only</td>
<td>Copy directory structure only, do not copy non-directory files.</td>
</tr>
<tr>
<td>create_symlinks</td>
<td>Make symbolic links instead of copies of files. The source path shall be an absolute path unless the destination path is in the current directory.</td>
</tr>
<tr>
<td>create_hard_links</td>
<td>Make hard links instead of copies of files.</td>
</tr>
</tbody>
</table>

Table 131: Enum class perms

<table>
<thead>
<tr>
<th>Name</th>
<th>Value (octal)</th>
<th>POSIX macro</th>
<th>Definition or notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0</td>
<td></td>
<td>There are no permissions set for the file.</td>
</tr>
<tr>
<td>owner_read</td>
<td>0400</td>
<td>S_IRUSR</td>
<td>Read permission, owner</td>
</tr>
<tr>
<td>owner_write</td>
<td>0200</td>
<td>S_IWUSR</td>
<td>Write permission, owner</td>
</tr>
<tr>
<td>owner_exec</td>
<td>0100</td>
<td>S_IXUSR</td>
<td>Execute/search permission, owner</td>
</tr>
<tr>
<td>owner_all</td>
<td>0700</td>
<td>S_IRWXU</td>
<td>Read, write, execute/search by owner; owner_read</td>
</tr>
<tr>
<td>group_read</td>
<td>040</td>
<td>S_IRGRP</td>
<td>Read permission, group</td>
</tr>
<tr>
<td>group_write</td>
<td>020</td>
<td>S_IWGRP</td>
<td>Write permission, group</td>
</tr>
<tr>
<td>group_exec</td>
<td>010</td>
<td>S_IXGRP</td>
<td>Execute/search permission, group</td>
</tr>
<tr>
<td>group_all</td>
<td>070</td>
<td>S_IRWXG</td>
<td>Read, write, execute/search by group; group_read</td>
</tr>
<tr>
<td>others_read</td>
<td>04</td>
<td>S_IROTH</td>
<td>Read permission, others</td>
</tr>
<tr>
<td>others_write</td>
<td>02</td>
<td>S_IWOTH</td>
<td>Write permission, others</td>
</tr>
<tr>
<td>others_exec</td>
<td>01</td>
<td>S_IXOTH</td>
<td>Execute/search permission, others</td>
</tr>
<tr>
<td>others_all</td>
<td>07</td>
<td>S_IRWXO</td>
<td>Read, write, execute/search by others; others_read</td>
</tr>
<tr>
<td>all</td>
<td>0777</td>
<td></td>
<td>owner_all</td>
</tr>
<tr>
<td>set_uid</td>
<td>04000</td>
<td>S_ISUID</td>
<td>Set-user-ID on execution</td>
</tr>
<tr>
<td>set_gid</td>
<td>02000</td>
<td>S_ISGID</td>
<td>Set-group-ID on execution</td>
</tr>
<tr>
<td>sticky_bit</td>
<td>01000</td>
<td>S_ISVTX</td>
<td>Operating system dependent.</td>
</tr>
<tr>
<td>mask</td>
<td>07777</td>
<td>all</td>
<td>set_uid</td>
</tr>
<tr>
<td>unknown</td>
<td>0xFFFF</td>
<td></td>
<td>The permissions are not known, such as when a file_status object is created without specifying the permissions</td>
</tr>
</tbody>
</table>

§ 29.11.8.3
29.11.8.4 Enum class perms

The enum class type perms is a bitmask type (16.3.3.3.4) that specifies bitmask constants used to identify
file permissions, with the meanings listed in Table 131.

29.11.8.5 Enum class perm_options

The enum class type perm_options is a bitmask type (16.3.3.3.4) that specifies bitmask constants used
to control the semantics of permissions operations, with the meanings listed in Table 132. The bitmask
constants are bitmask elements. In Table 132 perm denotes a value of type perms passed to permissions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>permissions shall replace the file’s permission bits with perm</td>
</tr>
<tr>
<td>add</td>
<td>permissions shall replace the file’s permission bits with the bitwise OR of perm and the file’s current permission bits.</td>
</tr>
<tr>
<td>remove</td>
<td>permissions shall replace the file’s permission bits with the bitwise AND of the complement of perm and the file’s current permission bits.</td>
</tr>
<tr>
<td>nofollow</td>
<td>permissions shall change the permissions of a symbolic link itself rather than the permissions of the file the link resolves to.</td>
</tr>
</tbody>
</table>

29.11.8.6 Enum class directory_options

The enum class type directory_options is a bitmask type (16.3.3.3.4) that specifies bitmask constants
used to identify directory traversal options, with the meanings listed in Table 133. The constant none
represents the empty bitmask; every other constant in the table represents a distinct bitmask element.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Skip directory symlinks, permission denied is an error.</td>
</tr>
<tr>
<td>follow_directory_symlink</td>
<td>Follow rather than skip directory symlinks.</td>
</tr>
<tr>
<td>skip_permission_denied</td>
<td>Skip directories that would otherwise result in permission denied.</td>
</tr>
</tbody>
</table>

29.11.9 Class file_status

29.11.9.1 General

namespace std::filesystem {
  class file_status {
    public:
      // 29.11.9.2, constructors and destructor
      file_status() noexcept : file_status(file_type::none) {}
      explicit file_status(file_type ft,
        perms prms = perms::unknown) noexcept;
      file_status(const file_status&) noexcept = default;
      file_status(file_status&&) noexcept = default;
      ~file_status();

      // assignments
      file_status& operator=(const file_status&) noexcept = default;
      file_status& operator=(file_status&&) noexcept = default;

      // 29.11.9.4, modifiers
      void type(file_type ft) noexcept;
      void permissions(perms prms) noexcept;
  }
}
An object of type `file_status` stores information about the type and permissions of a file.

### 29.11.9.2 Constructors

```cpp
explicit file_status(file_type ft, perms prms = perms::unknown) noexcept;
```

**Postconditions:** type() == ft and permissions() == prms.

### 29.11.9.3 Observers

```cpp
file_type type() const noexcept;
perms permissions() const noexcept;
```

- **Returns:** The value of type() specified by the postconditions of the most recent call to a constructor, `operator=`, or `type(file_type)` function.
- **Returns:** The value of permissions() specified by the postconditions of the most recent call to a constructor, `operator=`, or `permissions(perms)` function.

### 29.11.9.4 Modifiers

```cpp
void type(file_type ft) noexcept;
void permissions(perms prms) noexcept;
```

- **Postconditions:** type() == ft.
- **Postconditions:** permissions() == prms.

## 29.11.10 Class directory_entry

### 29.11.10.1 General

```cpp
namespace std::filesystem {
    class directory_entry {
        public:
            // 29.11.10.2, constructors and destructor
directory_entry() noexcept = default;
directory_entry(const directory_entry&) = default;
directory_entry(directory_entry&&) noexcept = default;
explicit directory_entry(const filesystem::path& p);
directory_entry(const filesystem::path& p, error_code& ec);
~directory_entry();

            // assignments
directory_entry& operator=(const directory_entry&);
directory_entry& operator=(directory_entry&&) noexcept = default;

            // 29.11.10.3, modifiers
void assign(const filesystem::path& p);
void assign(const filesystem::path& p, error_code& ec);
void replace_filename(const filesystem::path& p);
void replace_filename(const filesystem::path& p, error_code& ec);
void refresh();
void refresh(error_code& ec) noexcept;

            // 29.11.10.4, observers
const filesystem::path& path() const noexcept;
operator const filesystem::path&() const noexcept;
bool exists() const;
```
bool exists(error_code& ec) const noexcept;
bool is_block_file() const;
bool is_block_file(error_code& ec) const noexcept;
bool is_character_file() const;
bool is_character_file(error_code& ec) const noexcept;
bool is_directory() const;
bool is_directory(error_code& ec) const noexcept;
bool is_fifo() const;
bool is_fifo(error_code& ec) const noexcept;
bool is_other() const;
bool is_other(error_code& ec) const noexcept;
bool is_regular_file() const;
bool is_regular_file(error_code& ec) const noexcept;
bool is_socket() const;
bool is_socket(error_code& ec) const noexcept;
bool is_symlink() const;
bool is_symlink(error_code& ec) const noexcept;
uintmax_t file_size() const;
uintmax_t file_size(error_code& ec) const noexcept;
uintmax_t hard_link_count() const;
uintmax_t hard_link_count(error_code& ec) const noexcept;
file_time_type last_write_time() const;
file_time_type last_write_time(error_code& ec) const noexcept;
file_status status() const;
file_status status(error_code& ec) const noexcept;
file_status symlink_status() const;
file_status symlink_status(error_code& ec) const noexcept;

bool operator==(const directory_entry& rhs) const noexcept;
strong_ordering operator<=>(const directory_entry& rhs) const noexcept;

private:
filesystem::path pathobject; // exposition only
friend class directory_iterator; // exposition only
};

A directory_entry object stores a path object and may store additional objects for file attributes such as
hard link count, status, symlink status, file size, and last write time.

Implementations should store such additional file attributes during directory iteration if their values are avail-
able and storing the values would allow the implementation to eliminate file system accesses by directory-
entry observer functions (29.11.13). Such stored file attribute values are said to be cached.

[Note 1: For purposes of exposition, class directory_iterator (29.11.11) is shown above as a friend of class
directory_entry. Friendship allows the directory_iterator implementation to cache already available attribute
values directly into a directory_entry object without the cost of an unneeded call to refresh(). — end note]

[Example 1:

using namespace std::filesystem;

// use possibly cached last write time to minimize disk accesses
for (auto& x : directory_iterator("."))
{
    std::cout << x.path() << " " << x.last_write_time() << std::endl;
}

// call refresh() to refresh a stale cache
for (auto& x : directory_iterator("."))
{
    lengthy_function(x.path()); // cache becomes stale
    x.refresh();
    std::cout << x.path() << " " << x.last_write_time() << std::endl;
}
On implementations that do not cache the last write time, both loops will result in a potentially expensive call to the `std::filesystem::last_write_time` function. On implementations that do cache the last write time, the first loop will use the cached value and so will not result in a potentially expensive call to the `std::filesystem::last_write_time` function. The code is portable to any implementation, regardless of whether or not it employs caching. — end example

### 29.11.10.2 Constructors

```cpp
explicit directory_entry(const filesystem::path& p);
directory_entry(const filesystem::path& p, error_code& ec);
```

1. **Effects**: Calls `refresh()` or `refresh(ec)`, respectively.
2. **Postconditions**: `path() == p` if no error occurs, otherwise `path() == filesystem::path()`.
3. **Throws**: As specified in 29.11.5.

### 29.11.10.3 Modifiers

```cpp
void assign(const filesystem::path& p);
void assign(const filesystem::path& p, error_code& ec);
```

1. **Effects**: Equivalent to `pathobject = p`, then `refresh()` or `refresh(ec)`, respectively. If an error occurs, the values of any cached attributes are unspecified.
2. **Throws**: As specified in 29.11.5.

```cpp
void replace_filename(const filesystem::path& p);
void replace_filename(const filesystem::path& p, error_code& ec);
```

3. **Effects**: Equivalent to `pathobject.replace_filename(p)`, then `refresh()` or `refresh(ec)`, respectively. If an error occurs, the values of any cached attributes are unspecified.
4. **Throws**: As specified in 29.11.5.

```cpp
void refresh();
void refresh(error_code& ec) noexcept;
```

5. **Effects**: Stores the current values of any cached attributes of the file `p` resolves to. If an error occurs, an error is reported (29.11.5) and the values of any cached attributes are unspecified.
6. **Throws**: As specified in 29.11.5.

### 29.11.10.4 Observers

Unqualified function names in the **Returns**: elements of the `directory_entry` observers described below refer to members of the `std::filesystem` namespace.

```cpp
const filesystem::path& path() const noexcept;
operator const filesystem::path&() const noexcept;
```

2. **Returns**: `pathobject`.

```cpp
bool exists() const;
bool exists(error_code& ec) const noexcept;
```

3. **Returns**: `exists(this->status())` or `exists(this->status(ec))`, respectively.
4. **Throws**: As specified in 29.11.5.

```cpp
bool is_block_file() const;
bool is_block_file(error_code& ec) const noexcept;
```

5. **Returns**: `is_block_file(this->status())` or `is_block_file(this->status(ec))`, respectively.
6. **Throws**: As specified in 29.11.5.

```cpp
bool is_character_file() const;
bool is_character_file(error_code& ec) const noexcept;
```

7. **Returns**: `is_character_file(this->status())` or `is_character_file(this->status(ec))`, respectively.
§ 29.11.10.4 1485

8  \textit{Throws:} As specified in \ref{29.11.5}.

bool is_directory() const;
bool is_directory(error_code& ec) const noexcept;

9  \textit{Returns:} is_directory(this->status()) or is_directory(this->status(ec)), respectively.
8  \textit{Throws:} As specified in \ref{29.11.5}.

bool is_fifo() const;
bool is_fifo(error_code& ec) const noexcept;

10 \textit{Returns:} is_fifo(this->status()) or is_fifo(this->status(ec)), respectively.
9  \textit{Throws:} As specified in \ref{29.11.5}.

bool is_other() const;
bool is_other(error_code& ec) const noexcept;

11 \textit{Returns:} is_other(this->status()) or is_other(this->status(ec)), respectively.
10 \textit{Throws:} As specified in \ref{29.11.5}.

bool is_regular_file() const;
bool is_regular_file(error_code& ec) const noexcept;

12 \textit{Returns:} is_regular_file(this->status()) or is_regular_file(this->status(ec)), respectively.
11 \textit{Throws:} As specified in \ref{29.11.5}.

bool is_socket() const;
bool is_socket(error_code& ec) const noexcept;

13 \textit{Returns:} is_socket(this->status()) or is_socket(this->status(ec)), respectively.
12 \textit{Throws:} As specified in \ref{29.11.5}.

bool is_symlink() const;
bool is_symlink(error_code& ec) const noexcept;

14 \textit{Returns:} is_symlink(this->symlink_status()) or is_symlink(this->symlink_status(ec)), respectively.
13 \textit{Throws:} As specified in \ref{29.11.5}.

uintmax_t file_size() const;
uintmax_t file_size(error_code& ec) const noexcept;

15 \textit{Returns:} If cached, the file size attribute value. Otherwise, file_size(path()) or file_size(path(), ec), respectively.
14 \textit{Throws:} As specified in \ref{29.11.5}.

uintmax_t hard_link_count() const;
uintmax_t hard_link_count(error_code& ec) const noexcept;

16 \textit{Returns:} If cached, the hard link count attribute value. Otherwise, hard_link_count(path()) or hard_link_count(path(), ec), respectively.
15 \textit{Throws:} As specified in \ref{29.11.5}.

file_time_type last_write_time() const;
file_time_type last_write_time(error_code& ec) const noexcept;

17 \textit{Returns:} If cached, the last write time attribute value. Otherwise, last_write_time(path()) or last_write_time(path(), ec), respectively.
16 \textit{Throws:} As specified in \ref{29.11.5}.

file_status status() const;
file_status status(error_code& ec) const noexcept;

18 \textit{Returns:} If cached, the status attribute value. Otherwise, status(path()) or status(path(), ec), respectively.
28   Throws: As specified in 29.11.5.

    file_status symlink_status() const;
    file_status symlink_status(error_code& ec) const noexcept;

29   Returns: If cached, the symlink status attribute value. Otherwise, symlink_status(path()) or
            symlink_status(path(), ec), respectively.

30   Throws: As specified in 29.11.5.

    bool operator==(const directory_entry& rhs) const noexcept;

31   Returns: pathobject == rhs.pathobject.

32   Returns: pathobject <=> rhs.pathobject.

29.11.11 Class directory_iterator

29.11.11.1 General

1   An object of type directory_iterator provides an iterator for a sequence of directory_entry elements
    representing the path and any cached attribute values (29.11.10) for each file in a directory or in an
    implementation-defined directory-like file type.

[Note 1: For iteration into sub-directories, see class recursive_directory_iterator (29.11.12). — end note]

namespace std::filesystem {
    class directory_iterator {
        public:
            using iterator_category = input_iterator_tag;
            using value_type = directory_entry;
            using difference_type = ptrdiff_t;
            using pointer = const directory_entry*;
            using reference = const directory_entry&;

            // 29.11.11.2, member functions
            directory_iterator() noexcept;
            explicit directory_iterator(const path& p);
            directory_iterator(const path& p, directory_options options);
            directory_iterator(const path& p, error_code& ec);
            directory_iterator(const path& p, directory_options options,
                                error_code& ec);
            directory_iterator(const directory_iterator& rhs);
            directory_iterator(directory_iterator&& rhs) noexcept;
            ~directory_iterator();

            directory_iterator& operator=(const directory_iterator& rhs);
            directory_iterator& operator=(directory_iterator&& rhs) noexcept;

            const directory_entry& operator*() const;
            const directory_entry* operator->() const;
            directory_iterator& operator++();
            directory_iterator& increment(error_code& ec);

            // other members as required by 23.3.5.3, input iterators
    };
}

2   directory_iterator meets the Cpp17InputIterator requirements (23.3.5.3).

3   If an iterator of type directory_iterator reports an error or is advanced past the last directory element,
    that iterator shall become equal to the end iterator value. The directory_iterator default constructor
    shall create an iterator equal to the end iterator value, and this shall be the only valid iterator for the end
    condition.

4   The end iterator is not dereferenceable.

5   Two end iterators are always equal. An end iterator shall not be equal to a non-end iterator.

§ 29.11.11.1
The result of calling the \texttt{path()} member of the \texttt{directory_entry} object obtained by dereferencing a \texttt{directory_iterator} is a reference to a \texttt{path} object composed of the directory argument from which the iterator was constructed with filename of the directory entry appended as if by \texttt{operator/=}.

Directory iteration shall not yield directory entries for the current (dot) and parent (dot-dot) directories.

The order of directory entries obtained by dereferencing successive increments of a \texttt{directory_iterator} is unspecified.

Constructors and non-const \texttt{directory_iterator} member functions store the values of any cached attributes (29.11.10) in the \texttt{directory_entry} element returned by \texttt{operator*()}. \texttt{directory_iterator} member functions shall not directly or indirectly call any \texttt{directory_entry refresh} function.

[Note 2: The exact mechanism for storing cached attribute values is not exposed to users. For exposition, class \texttt{directory_iterator} is shown in 29.11.10 as a friend of class \texttt{directory_entry}. — end note]

[Note 3: A path obtained by dereferencing a directory iterator might not actually exist; it could be a symbolic link to a non-existent file. Recursively walking directory trees for purposes of removing and renaming entries might invalidate symbolic links that are being followed. — end note]

[Note 4: If a file is removed from or added to a directory after the construction of a \texttt{directory_iterator} for the directory, it is unspecified whether or not subsequently incrementing the iterator will ever result in an iterator referencing the removed or added directory entry. See POSIX \texttt{readdir_r}. — end note]

29.11.11.2 Members

\begin{verbatim}
directory_iterator() noexcept;
\end{verbatim}  
\textbf{Effects:} Constructs the end iterator.

\begin{verbatim}
explicit directory_iterator(const path& p);
directory_iterator(const path& p, directory_options options);
directory_iterator(const path& p, error_code& ec);
directory_iterator(const path& p, directory_options options, error_code& ec);
\end{verbatim}  
\textbf{Effects:} For the directory that \( p \) resolves to, constructs an iterator for the first element in a sequence of \texttt{directory_entry} elements representing the files in the directory, if any; otherwise the end iterator. However, if

\begin{verbatim}
(options & directory_options::skip_permission_denied) != directory_options::none
\end{verbatim}  
and construction encounters an error indicating that permission to access \( p \) is denied, constructs the end iterator and does not report an error.

\begin{verbatim}
directory_iterator(const directory_iterator& rhs);
directory_iterator(directory_iterator&& rhs) noexcept;
\end{verbatim}  
\textbf{Postconditions:} \texttt{*this} has the original value of \texttt{rhs}.

\begin{verbatim}
directory_iterator& operator=(const directory_iterator& rhs);
directory_iterator& operator=(directory_iterator&& rhs) noexcept;
\end{verbatim}  
\textbf{Effects:} If \texttt{*this} and \texttt{rhs} are the same object, the member has no effect.

\begin{verbatim}
Postconditions: \texttt{*this} has the original value of \texttt{rhs}.
\end{verbatim}  
\textbf{Returns:} \texttt{*this}.

\begin{verbatim}
directory_iterator& operator++();
directory_iterator& increment(error_code& ec);
\end{verbatim}  
\textbf{Effects:} As specified for the prefix increment operation of Input iterators (23.3.5.3).

\begin{verbatim}
Returns: \texttt{*this}.
\end{verbatim}  
\textbf{Throws:} As specified in 29.11.5.

29.11.11.3 Non-member functions

These functions enable range access for \texttt{directory_iterator}. 

\begin{verbatim}
\end{verbatim}
directory_iterator begin(directory_iterator iter) noexcept;

Returns: iter.

directory_iterator end(const directory_iterator&) noexcept;

Returns: directory_iterator().

29.11.12 Class recursive_directory_iterator [fs.class.rec.dir.itr]

29.11.12.1 General [fs.class.rec.dir.itr.general]

An object of type recursive_directory_iterator provides an iterator for a sequence of directory_entry elements representing the files in a directory or in an implementation-defined directory-like file type, and its sub-directories.

namespace std::filesystem {

    class recursive_directory_iterator {

    public:
        using iterator_category = input_iterator_tag;
        using value_type = directory_entry;
        using difference_type = ptrdiff_t;
        using pointer = const directory_entry*;
        using reference = const directory_entry&;

        // 29.11.12.2, constructors and destructor
        recursive_directory_iterator() noexcept;
        explicit recursive_directory_iterator(const path& p);
        recursive_directory_iterator(const path& p, directory_options options);
        recursive_directory_iterator(const path& p, directory_options options,
                                     error_code& ec);
        recursive_directory_iterator(const path& p, error_code& ec);
        recursive_directory_iterator(const recursive_directory_iterator& rhs);
        recursive_directory_iterator(recursive_directory_iterator&& rhs) noexcept;
        ~recursive_directory_iterator();

        // 29.11.12.2, observers
        directory_options options() const;
        int depth() const;
        bool recursion_pending() const;

        const directory_entry& operator*() const;
        const directory_entry* operator->() const;

        // 29.11.12.2, modifiers
        recursive_directory_iterator&
            operator=(const recursive_directory_iterator& rhs);
        recursive_directory_iterator&
            operator=(recursive_directory_iterator&& rhs) noexcept;

        recursive_directory_iterator& operator++();
        recursive_directory_iterator& increment(error_code& ec);

        void pop();
        void pop(error_code& ec);
        void disable_recursion_pending();

        // other members as required by 23.3.5.3, input iterators
    }

};

2 Calling options, depth, recursion_pending, pop or disable_recursion_pending on an iterator that is not dereferenceable results in undefined behavior.

3 The behavior of a recursive_directory_iterator is the same as a directory_iterator unless otherwise specified.
29.11.12.2 Members

recursive_directory_iterator() noexcept;

Effects: Constructs the end iterator.

explicit recursive_directory_iterator(const path& p);
recursive_directory_iterator(const path& p, directory_options options);
recursive_directory_iterator(const path& p, directory_options options, error_code& ec);
recursive_directory_iterator(const path& p, error_code& ec);

Effects: Constructs an iterator representing the first entry in the directory to which p resolves, if any; otherwise, the end iterator. However, if

(options & directory_options::skip_permission_denied) != directory_options::none

and construction encounters an error indicating that permission to access p is denied, constructs the end iterator and does not report an error.

Postconditions: options() == options for the signatures with a directory_options argument, otherwise options() == directory_options::none.

Throws: As specified in 29.11.5.

[Note 1: Use recursive_directory_iterator(".") rather than recursive_directory_iterator("") to iterate over the current directory. —end note]

recursive_directory_iterator(const recursive_directory_iterator& rhs);

Postconditions:

(7.1) options() == rhs.options()
(7.2) depth() == rhs.depth()
(7.3) recursion_pending() == rhs.recursion_pending()

recursive_directory_iterator(recursive_directory_iterator&& rhs) noexcept;

Postconditions: options(), depth(), and recursion_pending() have the values that rhs.options(), rhs.depth(), and rhs.recursion_pending(), respectively, had before the function call.

recursive_directory_iterator& operator=(const recursive_directory_iterator& rhs);

Effects: If *this and rhs are the same object, the member has no effect.

Postconditions:

(10.1) options() == rhs.options()
(10.2) depth() == rhs.depth()
(10.3) recursion_pending() == rhs.recursion_pending()

Returns: *this.

recursive_directory_iterator& operator=(recursive_directory_iterator&& rhs) noexcept;

Effects: If *this and rhs are the same object, the member has no effect.

Postconditions: options(), depth(), and recursion_pending() have the values that rhs.options(), rhs.depth(), and rhs.recursion_pending(), respectively, had before the function call.

Returns: *this.

directory_options options() const;

Returns: The value of the argument passed to the constructor for the options parameter, if present, otherwise directory_options::none.

Throws: Nothing.
int depth() const;
    
Returns: The current depth of the directory tree being traversed.

[Note 3: The initial directory is depth 0, its immediate subdirectories are depth 1, and so forth. — end note]

Throws: Nothing.

bool recursion_pending() const;
    
Returns: true if disable_recursion_pending() has not been called subsequent to the prior construction or increment operation, otherwise false.

Throws: Nothing.

recursive_directory_iterator& operator++();
recursive_directory_iterator& increment(error_code& ec);
    
Effects: As specified for the prefix increment operation of Input iterators (23.3.5.3), except that:

(21.1) — If there are no more entries at the current depth, then if depth() != 0 iteration over the parent directory resumes; otherwise *this = recursive_directory_iterator().

(21.2) — Otherwise if
    recursion_pending() && is_directory((*this)->status()) &&
    (!is_symlink((*this)->symlink_status()) ||
    (options() & directory_options::follow_directory_symlink) != directory_options::none)
then either directory (*this)->path() is recursively iterated into or, if

    (options() & directory_options::skip_permission_denied) != directory_options::none
and an error occurs indicating that permission to access directory (*this)->path() is denied,

then directory (*this)->path() is treated as an empty directory and no error is reported.

Returns: *this.

Throws: As specified in 29.11.5.

void pop();
void pop(error_code& ec);
    
Effects: If depth() == 0, set *this to recursive_directory_iterator(). Otherwise, cease iteration of the directory currently being iterated over, and continue iteration over the parent directory.

Throws: As specified in 29.11.5.

Remarks: Any copies of the previous value of *this are no longer required to be dereferenceable nor to be in the domain of ==.

void disable_recursion_pending();
    
Postconditions: recursion_pending() == false.

[Note 4: disable_recursion_pending() is used to prevent unwanted recursion into a directory. — end note]

29.11.12.3 Non-member functions

These functions enable use of recursive_directory_iterator with range-based for statements.

recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;
    
Returns: iter.

recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;
    
Returns: recursive_directory_iterator().

29.11.13 Filesystem operation functions

29.11.13.1 General

Filesystem operation functions query or modify files, including directories, in external storage.

[Note 1: Because hardware failures, network failures, file system races (29.11.2.4), and many other kinds of errors occur frequently in file system operations, any filesystem operation function, no matter how apparently innocuous, can encounter an error; see 29.11.5. — end note]
29.11.13.2 Absolute

path absolute(const path& p);
path absolute(const path& p, error_code& ec);

Effects: Composes an absolute path referencing the same file system location as p according to the operating system (29.11.2.3).

Returns: The composed path. The signature with argument ec returns path() if an error occurs.

[Note 1: For the returned path, rp, rp.is_absolute() is true unless an error occurs. — end note]

Throws: As specified in 29.11.5.

[Note 2: To resolve symlinks, or perform other sanitization which might require queries to secondary storage, such as hard disks, consider canonical (29.11.13.3). — end note]

[Note 3: Implementations are strongly encouraged to not query secondary storage, and not consider !exists(p) an error. — end note]

[Example 1: For POSIX-based operating systems, absolute(p) is simply current_path()/p. For Windows-based operating systems, absolute might have the same semantics as GetFullPathNameW. — end example]

29.11.13.3 Canonical

path canonical(const path& p);
path canonical(const path& p, error_code& ec);

Effects: Converts p to an absolute path that has no symbolic link, dot, or dot-dot elements in its pathname in the generic format.

Returns: A path that refers to the same file system object as absolute(p). The signature with argument ec returns path() if an error occurs.

Throws: As specified in 29.11.5.

Remarks: !exists(p) is an error.

29.11.13.4 Copy

void copy(const path& from, const path& to);

Effects: Equivalent to copy(from, to, copy_options::none).

void copy(const path& from, const path& to, error_code& ec);

Effects: Equivalent to copy(from, to, copy_options::none, ec).

void copy(const path& from, const path& to, copy_options options);
void copy(const path& from, const path& to, copy_options options, error_code& ec);

Preconditions: At most one element from each option group (29.11.8.3) is set in options.

Effects: Before the first use of f and t:

(4.1) If
- (options & copy_options::create_symlinks) != copy_options::none ||
- (options & copy_options::skip_symlinks) != copy_options::none

then auto f = symlink_status(from) and if needed auto t = symlink_status(to).

(4.2) Otherwise, if
- (options & copy_options::copy_symlinks) != copy_options::none

then auto f = symlink_status(from) and if needed auto t = status(to).

(4.3) Otherwise, auto f = status(from) and if needed auto t = status(to).

Effects are then as follows:

(4.4) If f.type() or t.type() is an implementation-defined file type (29.11.8.2), then the effects are implementation-defined.

(4.5) Otherwise, an error is reported as specified in 29.11.5 if:

(4.5.1) exists(f) is false, or
is_directory(f) && is_regular_file(t) is true.

Otherwise, if is_symlink(f), then:

- If (options & copy_options::skip_symlinks) != copy_options::none then return.
- Otherwise if !exists(t) || is_directory(t) && options & copy_options::create_symlinks) != copy_options::none then copy_symlink(from, to).
- Otherwise report an error as specified in 29.11.5.

Otherwise, if is_regular_file(f), then:

- If (options & copy_options::directories_only) != copy_options::none, then return.
- Otherwise, if (options & copy_options::create_symlinks) != copy_options::none, then create a symbolic link to the source file.
- Otherwise, if (options & copy_options::create_hard_links) != copy_options::none, then create a hard link to the source file.
- Otherwise, if is_directory(t), then copy_file(from, to/from.filename(), options).
- Otherwise, copy_file(from, to, options).

Otherwise, if

is_directory(f) && (options & copy_options::create_symlinks) != copy_options::none

then report an error with an error_code argument equal to make_error_code(errc::is_not_directory).

Otherwise, if

is_directory(f) && (options & copy_options::recursive) != copy_options::none || options == copy_options::none

then:

- If exists(t) is false, then create_directory(to, from).
- Then, iterate over the files in from, as if by

for (const directory_entry& x : directory_iterator(from))
  copy(x.path(), to/x.path().filename(),
       options | copy_options::in-recursive-copy);

where in-recursive-copy is a bitmask element of copy_options that is not one of the elements in 29.11.8.3.

Otherwise, for the signature with argument ec, ec.clear().

Otherwise, no effects.

5 Throws: As specified in 29.11.5.

6 Remarks: For the signature with argument ec, any library functions called by the implementation shall have an error_code argument if applicable.

7 [Example 1]: Given this directory structure:

```
/dir1
 file1
 file2
/dir2
 file3
```

Calling copy("/dir1", "/dir3") would result in:

```
/dir1
 file1
 file2
```
Alternatively, calling `copy("/dir1", "/dir3", copy_options::recursive)` would result in:

```
/dir1
 file1
 file2
 dir2
/file3
/dir3
 file1
 file2
 dir2
/file3
```

— end example

### 29.11.13.5 Copy file

```
bool copy_file(const path& from, const path& to);
bool copy_file(const path& from, const path& to, error_code& ec);
```

1. **Returns**: `copy_file(from, to, copy_options::none)` or `copy_file(from, to, copy_options::none, ec)`, respectively.
2. **Throws**: As specified in 29.11.5.

```
bool copy_file(const path& from, const path& to, copy_options options);
bool copy_file(const path& from, const path& to, copy_options options,
 error_code& ec);
```

3. **Preconditions**: At most one element from each option group (29.11.8.3) is set in `options`.
4. **Effects**: As follows:
   1. (4.1) Report an error as specified in 29.11.5 if:
      1. (4.1.1) `is_regular_file(from)` is `false`, or
      1. (4.1.2) `exists(to)` is `true` and `is_regular_file(to)` is `false`, or
      1. (4.1.3) `exists(to)` is `true` and `equivalent(from, to)` is `true`, or
      1. (4.1.4) `exists(to)` is `true` and
         ```
         (options & (copy_options::skip_existing |
         copy_options::overwrite_existing |
         copy_options::update_existing)) == copy_options::none
         ```
   2. (4.2) Otherwise, copy the contents and attributes of the file `from` resolves to, to the file `to` resolves to, if:
      1. (4.2.1) `exists(to)` is `false`, or
      1. (4.2.2) `(options & copy_options::overwrite_existing) != copy_options::none`, or
      1. (4.2.3) `(options & copy_options::update_existing) != copy_options::none` and `from` is more recent than `to`, determined as if by use of the `last_write_time` function (29.11.13.26).
   3. (4.3) Otherwise, no effects.
5. **Returns**: `true` if the `from` file was copied, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.
6. **Throws**: As specified in 29.11.5.
7. **Complexity**: At most one direct or indirect invocation of `status(to)`.

### 29.11.13.6 Copy symlink

```
void copy_symlink(const path& existing_symlink, const path& new_symlink);
```
void copy_symlink(const path& existing_symlink, const path& new_symlink, error_code& ec) noexcept;

1 Effects: Equivalent to function(read_symlink(existing_symlink), new_symlink) or function(read_symlink(existing_symlink, ec), new_symlink, ec), respectively, where in each case function is create_symlink or create_directory_symlink as appropriate.

2 Throws: As specified in 29.11.5.

29.11.13.7 Create directories

bool create_directories(const path& p);
bool create_directories(const path& p, error_code& ec);

1 Effects: Calls create_directory() for each element of p that does not exist.

2 Returns: true if a new directory was created for the directory p resolves to, otherwise false.

3 Throws: As specified in 29.11.5.

4 Complexity: $O(n)$ where n is the number of elements of p.

29.11.13.8 Create directory

bool create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;

1 Effects: Creates the directory p resolves to, as if by POSIX mkdir with a second argument of static_cast<int>(perms::all). If mkdir fails because p resolves to an existing directory, no error is reported. Otherwise on failure an error is reported.

2 Returns: true if a new directory was created, otherwise false.

3 Throws: As specified in 29.11.5.

4 Complexities $O(n)$ where n is the number of elements of p.

bool create_directory(const path& p, const path& existing_p);
bool create_directory(const path& p, const path& existing_p, error_code& ec) noexcept;

1 Effects: Creates the directory p resolves to, with attributes copied from directory existing_p. The set of attributes copied is operating system dependent. If mkdir fails because p resolves to an existing directory, no error is reported. Otherwise on failure an error is reported.

2 Returns: true if a new directory was created with attributes copied from directory existing_p, otherwise false.

3 Throws: As specified in 29.11.5.

4 [Note 1: For POSIX-based operating systems, the attributes are those copied by native API stat(existing_p.c_str(), &attributes_stat) followed by mkdir(p.c_str(), attributes_stat.st_mode). For Windows-based operating systems, the attributes are those copied by native API CreateDirectoryExW(existing_p.c_str(), p.c_str(), 0). — end note]

5 Returns: true if a new directory was created with attributes copied from directory existing_p, otherwise false.

6 Throws: As specified in 29.11.5.

29.11.13.9 Create directory symlink

void create_directory_symlink(const path& to, const path& new_symlink);
void create_directory_symlink(const path& to, const path& new_symlink, error_code& ec) noexcept;

1 Effects: Establishes the postcondition, as if by POSIX symlink().

2 Postconditions: new_symlink resolves to a symbolic link file that contains an unspecified representation of to.

3 Throws: As specified in 29.11.5.

4 [Note 1: Some operating systems require symlink creation to identify that the link is to a directory. Thus, create_symlink() (instead of create_directory_symlink()) cannot be used reliably to create directory symlinks. — end note]

5 [Note 2: Some operating systems do not support symbolic links at all or support them only for regular files. Some file systems (such as the FAT file system) do not support symbolic links regardless of the operating system. — end note]
29.11.13.10  Create hard link  

```c
void create_hard_link(const path& to, const path& new_hard_link);
void create_hard_link(const path& to, const path& new_hard_link,
    error_code& ec) noexcept;
```

1 Effects: Establishes the postcondition, as if by POSIX `link()`.

2 Postconditions:

\[(2.1) \quad \text{exists}(\text{to}) \&\& \text{exists}(\text{new_hard_link}) \&\& \text{equivalent}(\text{to}, \text{new_hard_link})\]

\[(2.2) \quad \text{The contents of the file or directory to resolves to are unchanged.}\]

3 Throws: As specified in 29.11.5.

4 [Note 1: Some operating systems do not support hard links at all or support them only for regular files. Some file systems (such as the FAT file system) do not support hard links regardless of the operating system. Some file systems limit the number of links per file. — end note]

29.11.13.11  Create symlink  

```c
void create_symlink(const path& to, const path& new_symlink);
void create_symlink(const path& to, const path& new_symlink,
    error_code& ec) noexcept;
```

1 Effects: Establishes the postcondition, as if by POSIX `symlink()`.

2 Postconditions: new_symlink resolves to a symbolic link file that contains an unspecified representation of to.

3 Throws: As specified in 29.11.5.

4 [Note 1: Some operating systems do not support symbolic links at all or support them only for regular files. Some file systems (such as the FAT file system) do not support symbolic links regardless of the operating system. — end note]

29.11.13.12  Current path  

```c
path current_path();
path current_path(error_code& ec);
```

1 Returns: The absolute path of the current working directory, whose pathname in the native format is obtained as if by POSIX `getcwd()` . The signature with argument ec returns `path()` if an error occurs.

2 Throws: As specified in 29.11.5.

3 Remarks: The current working directory is the directory, associated with the process, that is used as the starting location in pathname resolution for relative paths.

4 [Note 1: The `current_path()` name was chosen to emphasize that the returned value is a path, not just a single directory name. — end note]

5 [Note 2: The current path as returned by many operating systems is a dangerous global variable. It might be changed unexpectedly by third-party or system library functions, or by another thread. — end note]

```c
void current_path(const path& p);
void current_path(const path& p, error_code& ec) noexcept;
```

6 Effects: Establishes the postcondition, as if by POSIX `chdir()`.

7 Postconditions: equivalent(p, current_path()).

8 Throws: As specified in 29.11.5.

9 [Note 3: The current path for many operating systems is a dangerous global state. It might be changed unexpectedly by a third-party or system library functions, or by another thread. — end note]

29.11.13.13  Equivalent  

```c
bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2, error_code& ec) noexcept;
```

1 Two paths are considered to resolve to the same file system entity if two candidate entities reside on the same device at the same location.


[Note 1: On POSIX platforms, this is determined as if by the values of the POSIX `stat` class, obtained as if by `stat()` for the two paths, having equal `st_dev` values and equal `st_ino` values. End note]

Returns: `true`, if `p1` and `p2` resolve to the same file system entity, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.

Throws: As specified in 29.11.5.

Remarks: `!exists(p1) || !exists(p2)` is an error.

### 29.11.13.14 Exists

```cpp
bool exists(file_status s) noexcept;
```

**Returns:** `status_known(s) && s.type() != file_type::not_found`.

```cpp
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;
```

Let `s` be a `file_status`, determined as if by `status(p)` or `status(p, ec)`, respectively.

**Effects:** The signature with argument `ec` calls `ec.clear()` if `status_known(s)`.

**Returns:** `exists(s)`.

**Throws:** As specified in 29.11.5.

### 29.11.13.15 File size

```cpp
uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;
```

**Effects:** If `exists(p)` is `false`, an error is reported (29.11.5).

**Returns:**

1. (2.1) If `is_regular_file(p)`, the size in bytes of the file `p` resolves to, determined as if by the value of the POSIX `stat` class member `st_size` obtained as if by POSIX `stat()`.

2. (2.2) Otherwise, the result is implementation-defined.

The signature with argument `ec` returns `static_cast<uintmax_t>(-1)` if an error occurs.

**Throws:** As specified in 29.11.5.

### 29.11.13.16 Hard link count

```cpp
uintmax_t hard_link_count(const path& p);
uintmax_t hard_link_count(const path& p, error_code& ec) noexcept;
```

**Returns:** The number of hard links for `p`. The signature with argument `ec` returns `static_cast<uintmax_t>(-1)` if an error occurs.

**Throws:** As specified in 29.11.5.

### 29.11.13.17 Is block file

```cpp
bool is_block_file(file_status s) noexcept;
```

**Returns:** `s.type() == file_type::block`.

```cpp
bool is_block_file(const path& p);
bool is_block_file(const path& p, error_code& ec) noexcept;
```

**Returns:** `is_block_file(status(p))` or `is_block_file(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

**Throws:** As specified in 29.11.5.

### 29.11.13.18 Is character file

```cpp
bool is_character_file(file_status s) noexcept;
```

**Returns:** `s.type() == file_type::character`.

§ 29.11.13.18 1496
bool is_character_file(const path& p);
bool is_character_file(const path& p, error_code& ec) noexcept;

Returns: is_character_file(status(p)) or is_character_file(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

Throws: As specified in 29.11.5.

29.11.13.19 Is directory

bool is_directory(file_status s) noexcept;

Returns: s.type() == file_type::directory.

bool is_directory(const path& p);
bool is_directory(const path& p, error_code& ec) noexcept;

Returns: is_directory(status(p)) or is_directory(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

Throws: As specified in 29.11.5.

29.11.13.20 Is empty

bool is_empty(const path& p);
bool is_empty(const path& p, error_code& ec);

Effects:

(1.1) — Determine file_status s, as if by status(p) or status(p, ec), respectively.
(1.2) — For the signature with argument ec, return false if an error occurred.
(1.3) — Otherwise, if is_directory(s):
   (1.3.1) — Create a variable itr, as if by directory_iterator itr(p) or directory_iterator itr(p, ec), respectively.
   (1.3.2) — For the signature with argument ec, return false if an error occurred.
   (1.3.3) — Otherwise, return itr == directory_iterator().
(1.4) — Otherwise:
   (1.4.1) — Determine uintmax_t sz, as if by file_size(p) or file_size(p, ec), respectively.
   (1.4.2) — For the signature with argument ec, return false if an error occurred.
   (1.4.3) — Otherwise, return sz == 0.

Throws: As specified in 29.11.5.

29.11.13.21 Is fifo

bool is_fifo(file_status s) noexcept;

Returns: s.type() == file_type::fifo.

bool is_fifo(const path& p);
bool is_fifo(const path& p, error_code& ec) noexcept;

Returns: is_fifo(status(p)) or is_fifo(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

Throws: As specified in 29.11.5.

29.11.13.22 Is other

bool is_other(file_status s) noexcept;

Returns: exists(s) && !is_regular_file(s) && !is_directory(s) && !is_symlink(s).

bool is_other(const path& p);
bool is_other(const path& p, error_code& ec) noexcept;

Returns: is_other(status(p)) or is_other(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.
3  Throws: As specified in 29.11.5.

29.11.13.23  Is regular file

    bool is_regular_file(file_status s) noexcept;

    Returns: s.type() == file_type::regular.

    bool is_regular_file(const path& p);

    Returns: is_regular_file(status(p)).

    Throws: filesystem_error if status(p) would throw filesystem_error.

    bool is_regular_file(const path& p, error_code& ec) noexcept;

    Effects: Sets ec as if by status(p, ec).

    [Note 1: file_type::none, file_type::not_found and file_type::unknown cases set ec to error values. To distinguish between cases, call the status function directly. — end note]

    Returns: is_regular_file(status(p, ec)). Returns false if an error occurs.

29.11.13.24  Is socket

    bool is_socket(file_status s) noexcept;

    Returns: s.type() == file_type::socket.

    bool is_socket(const path& p);

    bool is_socket(const path& p, error_code& ec) noexcept;

    Returns: is_socket(status(p)) or is_socket(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

    Throws: As specified in 29.11.5.

29.11.13.25  Is symlink

    bool is_symlink(file_status s) noexcept;

    Returns: s.type() == file_type::symlink.

    bool is_symlink(const path& p);

    bool is_symlink(const path& p, error_code& ec) noexcept;

    Returns: is_symlink(symlink_status(p)) or is_symlink(symlink_status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

    Throws: As specified in 29.11.5.

29.11.13.26  Last write time

    file_time_type last_write_time(const path& p);

    file_time_type last_write_time(const path& p, error_code& ec) noexcept;

    Returns: The time of last data modification of p, determined as if by the value of the POSIX stat class member st_mtime obtained as if by POSIX stat(). The signature with argument ec returns file_time_type::min() if an error occurs.

    Throws: As specified in 29.11.5.

    void last_write_time(const path& p, file_time_type new_time);

    void last_write_time(const path& p, file_time_type new_time, error_code& ec) noexcept;

    Effects: Sets the time of last data modification of the file resolved to by p to new_time, as if by POSIX futimens().

    Throws: As specified in 29.11.5.

    [Note 1: A postcondition of last_write_time(p) == new_time is not specified since it might not hold for file systems with coarse time granularity. — end note]
29.11.13.27 Permissions

void permissions(const path& p, perms prms, perm_options opts=perm_options::replace);
void permissions(const path& p, perms prms, error_code& ec) noexcept;
void permissions(const path& p, perms prms, perm_options opts, error_code& ec);

1 Preconditions: Exactly one of the perm_options constants replace, add, or remove is present in opts.
2 Effects: Applies the action specified by opts to the file p resolves to, or to file p itself if p is a symbolic link and perm_options::nofollow is set in opts. The action is applied as if by POSIX fchmodat().
3 [Note 1: Conceptually permissions are viewed as bits, but the actual implementation can use some other mechanism. — end note]
4 Throws: As specified in 29.11.5.
5 Remarks: The second signature behaves as if it had an additional parameter perm_options opts with an argument of perm_options::replace.

29.11.13.28 Proximate

path proximate(const path& p, error_code& ec);
1 Returns: proximate(p, current_path(), ec).
2 Throws: As specified in 29.11.5.

path proximate(const path& p, const path& base = current_path());
path proximate(const path& p, const path& base, error_code& ec);
3 Returns: For the first form:
   weakly_canonical(p).lexically_proximate(weakly_canonical(base));
   For the second form:
   weakly_canonical(p, ec).lexically_proximate(weakly_canonical(base, ec));
   or path() at the first error occurrence, if any.
4 Throws: As specified in 29.11.5.

29.11.13.29 Read symlink

path read_symlink(const path& p);
path read_symlink(const path& p, error_code& ec);
1 Returns: If p resolves to a symbolic link, a path object containing the contents of that symbolic link. The signature with argument ec returns path() if an error occurs.
2 Throws: As specified in 29.11.5.
3 [Note 1: It is an error if p does not resolve to a symbolic link. — end note]

29.11.13.30 Relative

path relative(const path& p, error_code& ec);
1 Returns: relative(p, current_path(), ec).
2 Throws: As specified in 29.11.5.

path relative(const path& p, const path& base = current_path());
path relative(const path& p, const path& base, error_code& ec);
3 Returns: For the first form:
   weakly_canonical(p).lexically_relative(weakly_canonical(base));
   For the second form:
   weakly_canonical(p, ec).lexically_relative(weakly_canonical(base, ec));
   or path() at the first error occurrence, if any.
4 Throws: As specified in 29.11.5.
29.11.13.31  Remove

bool remove(const path& p);
bool remove(const path& p, error_code& ec) noexcept;

1  Effects: If exists(symlink_status(p, ec)), the file p is removed as if by POSIX remove().
   [Note 1: A symbolic link is itself removed, rather than the file it resolves to. — end note]
2  Postconditions: exists(symlink_status(p)) is false.
3  Returns: false if p did not exist, otherwise true. The signature with argument ec returns false if
          an error occurs.
4  Throws: As specified in 29.11.5.

29.11.13.32  Remove all

uintmax_t remove_all(const path& p);
uintmax_t remove_all(const path& p, error_code& ec);

1  Effects: Recursively deletes the contents of p if it exists, then deletes file p itself, as if by POSIX
          remove().
   [Note 1: A symbolic link is itself removed, rather than the file it resolves to. — end note]
2  Postconditions: exists(symlink_status(p)) is false.
3  Returns: The number of files removed. The signature with argument ec returns static_cast<
          uintmax_t>(-1) if an error occurs.
4  Throws: As specified in 29.11.5.

29.11.13.33  Rename

void rename(const path& old_p, const path& new_p);
void rename(const path& old_p, const path& new_p, error_code& ec) noexcept;

1  Effects: Renames old_p to new_p, as if by POSIX rename().
   [Note 1: 
     (1.1) — If old_p and new_p resolve to the same existing file, no action is taken.
     (1.2) — Otherwise, the rename can include the following effects:
     (1.2.1) — if new_p resolves to an existing non-directory file, new_p is removed; otherwise,
     (1.2.2) — if new_p resolves to an existing directory, new_p is removed if empty on POSIX compliant
               operating systems but might be an error on other operating systems.
           A symbolic link is itself renamed, rather than the file it resolves to. — end note]
2  Throws: As specified in 29.11.5.

29.11.13.34  Resize file

void resize_file(const path& p, uintmax_t new_size);
void resize_file(const path& p, uintmax_t new_size, error_code& ec) noexcept;

1  Effects: Causes the size that would be returned by file_size(p) to be equal to new_size, as if by
          POSIX truncate().
2  Throws: As specified in 29.11.5.

29.11.13.35  Space

space_info space(const path& p);
space_info space(const path& p, error_code& ec) noexcept;

1  Returns: An object of type space_info. The value of the space_info object is determined as if
          by using POSIX statvfs to obtain a POSIX struct statvfs, and then multiplying its f_blocks,
          f_bfree, and f_bavail members by its f_frsize member, and assigning the results to the
          capacity, free, and available members respectively. Any members for which the value cannot be determined
          shall be set to static_cast<uintmax_t>(-1). For the signature with argument ec, all members are set to
          static_cast<uintmax_t>(-1) if an error occurs.
29.11.13.36 Status

file_status status(const path& p);

**Effects:** As if:

```
error_code ec;
file_status result = status(p, ec);
if (result.type() == file_type::none)
  throw filesystem_error(implementation-supplied-message, p, ec);
return result;
```

**Returns:** See above.

**Throws:** filesystem_error.

*[Note 1: result values of file_status(file_type::not_found) and file_status(file_type::unknown) are not considered failures and do not cause an exception to be thrown. — end note]*

file_status status(const path& p, error_code& ec) noexcept;

**Effects:** If possible, determines the attributes of the file p resolves to, as if by using POSIX stat() to obtain a POSIX struct stat. If, during attribute determination, the underlying file system API reports an error, sets ec to indicate the specific error reported. Otherwise, ec.clear().

*[Note 2: This allows users to inspect the specifics of underlying API errors even when the value returned by status() is not file_status(file_type::none). — end note]*

Let prms denote the result of (m & perms::mask), where m is determined as if by converting the st_mode member of the obtained struct stat to the type perms.

**Returns:**

*(6.1)* If ec != error_code():

*(6.1.1)* If the specific error indicates that p cannot be resolved because some element of the path does not exist, returns file_status(file_type::not_found).

*(6.1.2)* Otherwise, if the specific error indicates that p can be resolved but the attributes cannot be determined, returns file_status(file_type::unknown).

*(6.1.3)* Otherwise, returns file_status(file_type::none).

*[Note 3: These semantics distinguish between p being known not to exist, p existing but not being able to determine its attributes, and there being an error that prevents even knowing if p exists. These distinctions are important to some use cases. — end note]*

*(6.2)* Otherwise,

*(6.2.1)* If the attributes indicate a regular file, as if by POSIX S_ISREG, returns file_status(file_type::regular, prms).

*[Note 4: file_type::regular implies appropriate <fstream> operations would succeed, assuming no hardware, permission, access, or file system race errors. Lack of file_type::regular does not necessarily imply <fstream> operations would fail on a directory. — end note]*

*(6.2.2)* Otherwise, if the attributes indicate a directory, as if by POSIX S_ISDIR, returns file_status(file_type::directory, prms).

*[Note 5: file_type::directory implies that calling directory_iterator(p) would succeed. — end note]*

*(6.2.3)* Otherwise, if the attributes indicate a block special file, as if by POSIX S_ISBLK, returns file_status(file_type::block, prms).

*(6.2.4)* Otherwise, if the attributes indicate a character special file, as if by POSIX S_ISCHR, returns file_status(file_type::character, prms).

*(6.2.5)* Otherwise, if the attributes indicate a fifo or pipe file, as if by POSIX S_ISFIFO, returns file_status(file_type::fifo, prms).
(6.2.6) — Otherwise, if the attributes indicate a socket, as if by POSIX S_ISSOCK, returns `file_status(file_type::socket, prms)`.

(6.2.7) — Otherwise, if the attributes indicate an implementation-defined file type (29.11.8.2), returns `file_status(file_type::A, prms)`, where A is the constant for the implementation-defined file type.

(6.2.8) — Otherwise, returns `file_status(file_type::unknown, prms)`.

**Remarks:** If a symbolic link is encountered during pathname resolution, pathname resolution continues using the contents of the symbolic link.

### 29.11.13.37 Status known

```cpp
def status_known(file_status s) noexcept;
```

**Returns:** `s.type() != file_type::none`.

### 29.11.13.38 Symlink status

```cpp
file_status symlink_status(const path& p);
file_status symlink_status(const path& p, error_code& ec) noexcept;
```

**Effects:** Same as `status()`, above, except that the attributes of `p` are determined as if by using POSIX `lstat()` to obtain a POSIX `struct stat`.

**Let** `prms` **denote the result of** `(m & perms::mask)`, where `m` **is determined as if by converting the** `st_mode` **member of the obtained** `struct stat` **to the type** `perms`.

**Returns:** Same as `status()`, above, except that if the attributes indicate a symbolic link, as if by POSIX S_ISLNK, returns `file_status(file_type::symlink, prms)`. The signature with argument `ec` returns `file_status(file_type::none)` if an error occurs.

**Throws:** As specified in 29.11.5.

**Remarks:** Pathname resolution terminates if `p` names a symbolic link.

### 29.11.13.39 Temporary directory path

```cpp
path temp_directory_path();
path temp_directory_path(error_code& ec);
```

**Effects:** If `exists(p)` is `false` or `is_directory(p)` is `false`, an error is reported (29.11.5).

**Returns:** The path `p`. The signature with argument `ec` returns `path()` if an error occurs.

**Throws:** As specified in 29.11.5.

**Example 1:** For POSIX-based operating systems, an implementation might return the path supplied by the first environment variable found in the list TMPDIR, TMP, TEMP, TEMPDIR, or if none of these are found, "/tmp".

For Windows-based operating systems, an implementation might return the path reported by the Windows `GetTempPath` API function. — end example

### 29.11.13.40 Weakly canonical

```cpp
path weakly_canonical(const path& p);
path weakly_canonical(const path& p, error_code& ec);
```

**Effects:** Using `status(p)` or `status(p, ec)`, respectively, to determine existence, return a path composed by `operator/=` from the result of calling `canonical()` with a path argument composed of the leading elements of `p` that exist, if any, followed by the elements of `p` that do not exist, if any. For the first form, `canonical()` is called without an `error_code` argument. For the second form, `canonical()` is called with `ec` as an `error_code` argument, and `path()` is returned at the first error occurrence, if any.

**Postconditions:** The returned path is in normal form (29.11.6.2).

**Returns:** `p` with symlinks resolved and the result normalized (29.11.6.2).

**Throws:** As specified in 29.11.5.
Remarks: Implementations should avoid unnecessary normalization such as when canonical has already been called on the entirety of \( p \).

29.12 C library files

29.12.1 Header <cstdio> synopsis

namespace std {
    using size_t = see 17.2.4;
    using FILE = see below;
    using fpos_t = see below;
}

#define NULL see 17.2.3
#define _IOFBF see below
#define _IOLBF see below
#define _IONBF see below
#define BUFSIZ see below
#define EOF see below
#define FOPEN_MAX see below
#define FILENAME_MAX see below
#define L_tmpnam see below
#define SEEK_CUR see below
#define SEEK_END see below
#define SEEK_SET see below
#define TMP_MAX see below
#define stcerr see below
#define stdin see below
#define stdout see below

namespace std {
    int remove(const char* filename);
    int rename(const char* old_p, const char* new_p);
    FILE* tmpfile();
    char* tmpnam(char* s);
    int fclose(FILE* stream);
    int fflush(FILE* stream);
    FILE* fopen(const char* filename, const char* mode);
    FILE* freopen(const char* filename, const char* mode, FILE* stream);
    void setbuf(FILE* stream, char* buf);
    int setvbuf(FILE* stream, const char* buf, int mode, size_t size);
    int fprintf(FILE* stream, const char* format, ...);
    int fscanf(FILE* stream, const char* format, ...);
    int printf(const char* format, ...);
    int scanf(const char* format, ...);
    int snprintf(char* s, size_t n, const char* format, ...);
    int sprintf(char* s, const char* format, ...);
    int sscanf(const char* s, const char* format, ...);
    int vfprintf(FILE* stream, const char* format, va_list arg);
    int vfscanf(FILE* stream, const char* format, va_list arg);
    int vprintf(const char* format, va_list arg);
    int vscanf(const char* format, va_list arg);
    int vsnprintf(char* s, size_t n, const char* format, va_list arg);
    int vsprintf(char* s, const char* format, va_list arg);
    int vsscanf(const char* s, const char* format, va_list arg);
    int fgetc(FILE* stream);
    char* fgets(char* s, int n, FILE* stream);
    int fputc(int c, FILE* stream);
    int fputs(const char* s, FILE* stream);
    int getc(FILE* stream);
    int getchar();
    int putc(int c, FILE* stream);
    int putchar(int c);
    int puts(const char* s);
    int ungetc(int c, FILE* stream);
}

§ 29.12.1
size_t fread(void* ptr, size_t size, size_t nmemb, FILE* stream);
size_t fwrite(const void* ptr, size_t size, size_t nmemb, FILE* stream);
int fgetpos(FILE* stream, fpos_t* pos);
int fseek(FILE* stream, long int offset, int whence);
int fsetpos(FILE* stream, const fpos_t* pos);
long int ftell(FILE* stream);
void rewind(FILE* stream);
void clearerr(FILE* stream);
int feof(FILE* stream);
int ferror(FILE* stream);
void perror(const char* s);

1 The contents and meaning of the header `<cstdio>` are the same as the C standard library header `<stdio.h>`.

2 Calls to the function `tmpnam` with an argument that is a null pointer value may introduce a data race (16.4.6.10) with other calls to `tmpnam` with an argument that is a null pointer value.

SEE ALSO: ISO C 7.21

29.12.2 Header `<cstdint>` synopsis

#include `<cstdint>` // see 17.4.2

namespace std {
    using imaxdiv_t = see below;
    intmax_t imaxabs(intmax_t j);
    imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
    intmax_t strtoimax(const char* nptr, char** endptr, int base);
    uintmax_t strtoumax(const char* nptr, char** endptr, int base);
    intmax_t wcstoimax(const wchar_t* nptr, wchar_t** endptr, int base);
    uintmax_t wcstoumax(const wchar_t* nptr, wchar_t** endptr, int base);
    intmax_t abs(intmax_t);
    // optional, see below
    imaxdiv_t div(intmax_t);
    // optional, see below
}

#define PRIdN see below
#define PRIiN see below
#define PRIoN see below
#define PRIuN see below
#define PRIxN see below
#define PRIXN see below
#define SCNdN see below
#define SCNiN see below
#define SCNoN see below
#define SCNuN see below
#define SCNxN see below
#define PRIdLEASTN see below
#define PRIiLEASTN see below
#define PRIoLEASTN see below
#define PRIuLEASTN see below
#define PRIxLEASTN see below
#define PRIXLEASTN see below
#define SCNdLEASTN see below
#define SCNiLEASTN see below
#define SCNoLEASTN see below
#define SCNuLEASTN see below
#define PRIdFASTN see below
#define PRIiFASTN see below
#define PRIoFASTN see below
#define PRIuFASTN see below
#define PRIxFASTN see below
#define PRIXFASTN see below
The contents and meaning of the header `<cinttypes>` are the same as the C standard library header `<inttypes.h>`, with the following changes:

(1.1) — The header `<cinttypes>` includes the header `<cstdint>` (17.4.2) instead of `<stdint.h>`, and

(1.2) — if and only if the type `intmax_t` designates an extended integer type (6.8.2), the following function signatures are added:

```c
intmax_t abs(intmax_t);
imaxdiv_t div(intmax_t, intmax_t);
```

which shall have the same semantics as the function signatures `intmax_t imaxabs(intmax_t)` and `imaxdiv_t imaxdiv(intmax_t, intmax_t)`, respectively.

See also: ISO C 7.8
30 Regular expressions library [re]

30.1 General [re.general]
1 This Clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

2 The following subclauses describe a basic regular expression class template and its traits that can handle char-like (21.1) template arguments, two specializations of this class template that handle sequences of char and wchar_t, a class template that holds the result of a regular expression match, a series of algorithms that allow a character sequence to be operated upon by a regular expression, and two iterator types for enumerating regular expression matches, as summarized in Table 134.

Table 134: Regular expressions library summary [tab:re.summary]

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>30.4</td>
<td>Constants</td>
</tr>
<tr>
<td>30.5</td>
<td>Exception type</td>
</tr>
<tr>
<td>30.6</td>
<td>Traits</td>
</tr>
<tr>
<td>30.7</td>
<td>Regular expression template</td>
</tr>
<tr>
<td>30.8</td>
<td>Submatches</td>
</tr>
<tr>
<td>30.9</td>
<td>Match results</td>
</tr>
<tr>
<td>30.10</td>
<td>Algorithms</td>
</tr>
<tr>
<td>30.11</td>
<td>Iterators</td>
</tr>
<tr>
<td>30.12</td>
<td>Grammar</td>
</tr>
</tbody>
</table>

30.2 Requirements [re.req]
1 This subclause defines requirements on classes representing regular expression traits.

[Note 1: The class template regex_traits, defined in 30.6, meets these requirements. — end note]

2 The class template basic_regex, defined in 30.7, needs a set of related types and functions to complete the definition of its semantics. These types and functions are provided as a set of member typedef-names and functions in the template parameter traits used by the basic_regex class template. This subclause defines the semantics of these members.

3 To specialize class template basic_regex for a character container CharT and its related regular expression traits class Traits, use basic_regex<CharT, Traits>.

4 In Table 135 X denotes a traits class defining types and functions for the character container type charT; u is an object of type X; v is an object of type const X; p is a value of type const charT*; I1 and I2 are input iterators (23.3.5.3); F1 and F2 are forward iterators (23.3.5.5); c is a value of type const charT; s is an object of type X::string_type; cs is an object of type const X::string_type; b is a value of type bool; I is a value of type int; cl is an object of type X::char_class_type, and loc is an object of type X::locale_type.

Table 135: Regular expression traits class requirements [tab:re.req]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::char_type</td>
<td>charT</td>
<td>The character container type used in the implementation of class template basic_regex.</td>
</tr>
<tr>
<td>X::string_type</td>
<td>basic_string&lt;CharT&gt;</td>
<td></td>
</tr>
<tr>
<td>X::locale_type</td>
<td>A copy constructible type</td>
<td>A type that represents the locale used by the traits class.</td>
</tr>
</tbody>
</table>
Table 135: Regular expression traits class requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::char_class_type</td>
<td>A bitmask type</td>
<td>A bitmask type representing a particular character classification.</td>
</tr>
<tr>
<td></td>
<td>type (16.3.3.3.4).</td>
<td></td>
</tr>
<tr>
<td>X::length(p)</td>
<td>size_t</td>
<td>Yields the smallest i such that p[i] == 0. Complexity is linear in i.</td>
</tr>
<tr>
<td>v.translate(c)</td>
<td>X::char_type</td>
<td>Returns a character such that for any character d that is to be considered equivalent to c then v.translate(c) == v.translate(d).</td>
</tr>
<tr>
<td>v.translate_nocase(c)</td>
<td>X::char_type</td>
<td>For all characters C that are to be considered equivalent to c when comparisons are to be performed without regard to case, then v.translate_nocase(c) == v.translate_nocase(C).</td>
</tr>
<tr>
<td>v.transform(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) then v.transform(G1, G2) &lt; v.transform(H1, H2).</td>
</tr>
<tr>
<td>v.transform_primary(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) when character case is not considered then v.transform_primary(G1, G2) &lt; v.transform_primary(H1, H2).</td>
</tr>
<tr>
<td>v.lookup_collatename(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range [F1, F2). Returns an empty string if the character sequence is not a valid collating element.</td>
</tr>
<tr>
<td>v.lookup_classname(F1, F2, b)</td>
<td>X::char_class_type</td>
<td>Converts the character sequence designated by the iterator range [F1, F2) into a value of a bitmask type that can subsequently be passed to isctype. Values returned from lookup_classname can be bitwise OR’ed together; the resulting value represents membership in either of the corresponding character classes. If b is true, the returned bitmask is suitable for matching characters without regard to their case. Returns 0 if the character sequence is not the name of a character class recognized by X. The value returned shall be independent of the case of the characters in the sequence.</td>
</tr>
<tr>
<td>v.isctype(c, cl)</td>
<td>bool</td>
<td>Returns true if character c is a member of one of the character classes designated by cl, false otherwise.</td>
</tr>
<tr>
<td>v.value(c, I)</td>
<td>int</td>
<td>Returns the value represented by the digit c in base I if the character c is a valid digit in base I; otherwise returns -1. [Note 2: The value of I will only be 8, 10, or 16. —end note]</td>
</tr>
</tbody>
</table>
Table 135: Regular expression traits class requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>u.imbue(loc)</td>
<td>X::locale_type</td>
<td>Imbues u with the locale loc and returns the previous locale used by u if any.</td>
</tr>
<tr>
<td>v.getloc()</td>
<td>X::locale_type</td>
<td>Returns the current locale used by v, if any.</td>
</tr>
</tbody>
</table>

5 [Note 3: Class template regex_traits meets the requirements for a regular expression traits class when it is specialized for char or wchar_t. This class template is described in the header <regex>, and is described in 30.6. — end note]

30.3 Header <regex> synopsis

```cpp
#include <compare>    // see 17.11.1
#include <initializer_list>    // see 17.10.2
namespace std {
    // 30.4, regex constants
    namespace regex_constants {
        using syntax_option_type = T1;
        using match_flag_type = T2;
        using error_type = T3;
    }
    // 30.5, class regex_error
    class regex_error;
    // 30.6, class template regex_traits
    template<class charT> struct regex_traits;
    // 30.7, class template basic_regex
    template<class charT, class traits = regex_traits<charT>> class basic_regex;
        using regex = basic_regex<char>;
        using wregex = basic_regex<wchar_t>;
    // 30.7.6, basic_regex swap
    template<class charT, class traits>
        void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);
    // 30.8, class template sub_match
    template<class BidirectionalIterator>
        class sub_match;
        using csub_match = sub_match<const char*>;
        using wcsub_match = sub_match<const wchar_t*>;
        using ssub_match = sub_match<string::const_iterator>;
        using wssub_match = sub_match<wstring::const_iterator>;
    // 30.8.3, sub_match non-member operators
    template<class BiIter>
        bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
    template<class BiIter>
        auto operator<=>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
    // 30.8.3, sub_match non-member operators
    template<class BiIter, class ST, class SA>
        bool operator==(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
    template<class BiIter, class ST, class SA>
        auto operator<=>(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```
template<class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
        const typename iterator_traits<BiIter>::value_type* rhs);

template<class BiIter>
auto operator<=>(const sub_match<BiIter>& lhs,
        const typename iterator_traits<BiIter>::value_type* rhs);

template<class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
        const typename iterator_traits<BiIter>::value_type& rhs);

template<class BiIter>
auto operator<=>(const sub_match<BiIter>& lhs,
        const typename iterator_traits<BiIter>::value_type& rhs);

template<class charT, class ST, class BiIter>
basic_ostream<charT, ST>&
operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

// 30.9, class template match_results
template<class BidirectionalIterator, class Allocator = allocator<sub_match<BidirectionalIterator>>> class match_results;

using cmatch = match_results<const char*>;
using wcmatch = match_results<const wchar_t*>;
using smatch = match_results<string::const_iterator>;
using wsmatch = match_results<wstring::const_iterator>;

// match_results comparisons
template<class BidirectionalIterator, class Allocator>
bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
        const match_results<BidirectionalIterator, Allocator>& m2);

// 30.9.8, match_results swap
template<class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
        match_results<BidirectionalIterator, Allocator>& m2);

// 30.10.2, function template regex_match
template<class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
        match_results<BidirectionalIterator, Allocator>& m,
        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags = regex_constants::match_default);

template<class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags = regex_constants::match_default);

template<class charT, class Allocator, class traits>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags = regex_constants::match_default);

template<class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
        match_results<typename basic_string<charT, ST, SA>::const_iterator,
        Allocator>& m,
        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags = regex_constants::match_default);

template<class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>&&,
        match_results<typename basic_string<charT, ST, SA>::const_iterator,
        Allocator> &,
        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags = regex_constants::match_default) = delete;
template<class charT, class traits>
bool regex_match(const charT* str,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

bool regex_match(const basic_string<charT, ST, SA>& s,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

// 30.10.3, function template regex_search
template<class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
    match_results<BidirectionalIterator, Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

bool regex_search(const charT* str,
    match_results<const charT*, Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

bool regex_search(const charT* str,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

bool regex_search(const basic_string<charT, ST, SA>& s,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

bool regex_search(const basic_string<charT, ST, SA>&&,
    match_results<typename basic_string<charT, ST, SA>::const_iterator,
        Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default) = delete;

// 30.10.4, function template regex_replace
template<class OutputIterator, class BidirectionalIterator, class traits, class charT, class ST, class SA>
OutputIterator
    regex_replace(OutputIterator out,
        BidirectionalIterator first, BidirectionalIterator last,
        const basic_regex<charT, traits>& e,
        const basic_string<charT, ST, SA>& fmt,
        regex_constants::match_flag_type flags = regex_constants::match_default);

OutputIterator
    regex_replace(OutputIterator out,
        BidirectionalIterator first, BidirectionalIterator last,
        const charT* fmt,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template<class traits, class charT, class ST, class SA, class FST, class FSA>
basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
const basic_string<charT, FST, FSA>& fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);

template<class traits, class charT, class ST, class SA>
basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);

template<class traits, class charT, class ST, class SA>
basic_string<charT>
regex_replace(const charT* s,
const basic_regex<charT, traits>& e,
const basic_string<charT, ST, SA>& fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);

template<class traits, class charT>
basic_string<charT>
regex_replace(const charT* s,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);

// 30.11.1, class template regex_iterator
template<class BidirectionalIterator,
class charT = typename iterator_traits<BidirectionalIterator>::value_type,
class traits = regex_traits<charT>>
class regex_iterator;

using cregex_iterator = regex_iterator<const char*>;
using wcregex_iterator = regex_iterator<const wchar_t*>;
using sregex_iterator = regex_iterator<string::const_iterator>
using wsregex_iterator = regex_iterator<wstring::const_iterator>;

// 30.11.2, class template regex_token_iterator
template<class BidirectionalIterator,
class charT = typename iterator_traits<BidirectionalIterator>::value_type,
class traits = regex_traits<charT>>
class regex_token_iterator;

using cregex_token_iterator = regex_token_iterator<const char*>;
using wcregex_token_iterator = regex_token_iterator<const wchar_t*>;
using sregex_token_iterator = regex_token_iterator<string::const_iterator>;
using wsregex_token_iterator = regex_token_iterator<wstring::const_iterator>;

namespace pmr {
  template<class BidirectionalIterator>
  using match_results =
  std::match_results<BidirectionalIterator,
  polymorphic_allocator<sub_match<BidirectionalIterator>>;)

  using cmatch = match_results<const char*>;
  using wcmatch = match_results<const wchar_t*>;
  using smatch = match_results<string::const_iterator>;
  using wsmatch = match_results<wstring::const_iterator>;
}

§ 30.3
30.4 Namespace std::regex_constants

30.4.1 General

The namespace std::regex_constants holds symbolic constants used by the regular expression library. This namespace provides three types, syntax_option_type, match_flag_type, and error_type, along with several constants of these types.

30.4.2 Bitmask type syntax_option_type

namespace std::regex_constants {
    using syntax_option_type = T1;
    inline constexpr syntax_option_type icase = unspecified;
    inline constexpr syntax_option_type nosubs = unspecified;
    inline constexpr syntax_option_type optimize = unspecified;
    inline constexpr syntax_option_type collate = unspecified;
    inline constexpr syntax_option_type ECMAScript = unspecified;
    inline constexpr syntax_option_type basic = unspecified;
    inline constexpr syntax_option_type extended = unspecified;
    inline constexpr syntax_option_type awk = unspecified;
    inline constexpr syntax_option_type grep = unspecified;
    inline constexpr syntax_option_type egrep = unspecified;
    inline constexpr syntax_option_type multiline = unspecified;
}

1 The type syntax_option_type is an implementation-defined bitmask type (16.3.3.3.4). Setting its elements has the effects listed in Table 136. A valid value of type syntax_option_type shall have at most one of the grammar elements ECMAScript, basic, extended, awk, grep, egrep, set. If no grammar element is set, the default grammar is ECMAScript.

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>icase</td>
<td>Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.</td>
</tr>
<tr>
<td>nosubs</td>
<td>Specifies that no sub-expressions shall be considered to be marked, so that when a regular expression is matched against a character container sequence, no sub-expression matches shall be stored in the supplied match_results object.</td>
</tr>
<tr>
<td>optimize</td>
<td>Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output.</td>
</tr>
<tr>
<td>collate</td>
<td>Specifies that character ranges of the form &quot;[a-b]&quot; shall be locale sensitive.</td>
</tr>
<tr>
<td>ECMAScript</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by ECMAScript in ECMA-262, as modified in 30.12. See also: ECMA-262 15.10</td>
</tr>
<tr>
<td>basic</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by basic regular expressions in POSIX. See also: POSIX, Base Definitions and Headers, Section 9.3</td>
</tr>
<tr>
<td>extended</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by extended regular expressions in POSIX. See also: POSIX, Base Definitions and Headers, Section 9.4</td>
</tr>
<tr>
<td>awk</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by the utility awk in POSIX.</td>
</tr>
<tr>
<td>grep</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by the utility grep in POSIX.</td>
</tr>
<tr>
<td>egrep</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by the utility grep when given the -E option in POSIX.</td>
</tr>
<tr>
<td>multiline</td>
<td>Specifies that ^ shall match the beginning of a line and $ shall match the end of a line, if the ECMAScript engine is selected.</td>
</tr>
</tbody>
</table>
30.4.3 Bitmask type match_flag_type

```cpp
namespace std::regex_constants {
    using match_flag_type = T2;
    inline constexpr match_flag_type match_default = {};
    inline constexpr match_flag_type match_not_bol = unspecified;
    inline constexpr match_flag_type match_not_eol = unspecified;
    inline constexpr match_flag_type match_not_bow = unspecified;
    inline constexpr match_flag_type match_not_eow = unspecified;
    inline constexpr match_flag_type match_any = unspecified;
    inline constexpr match_flag_type match_not_null = unspecified;
    inline constexpr match_flag_type match_continuous = unspecified;
    inline constexpr match_flag_type match_prev_avail = unspecified;
    inline constexpr match_flag_type format_default = {};
    inline constexpr match_flag_type format_sed = unspecified;
    inline constexpr match_flag_type format_no_copy = unspecified;
    inline constexpr match_flag_type format_first_only = unspecified;
}
```

1 The type `match_flag_type` is an implementation-defined bitmask type (16.3.3.3.4). The constants of that type, except for `match_default` and `format_default`, are bitmask elements. The `match_default` and `format_default` constants are empty bitmasks. Matching a regular expression against a sequence of characters `[first, last)` proceeds according to the rules of the grammar specified for the regular expression object, modified according to the effects listed in Table 137 for any bitmask elements set.

Table 137: `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first, last)`.

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
</table>
| `match_not_bol`  | The first character in the sequence `[first, last)` shall be treated as though it is not at the beginning of a line, so the character `^` in the regular expression shall not match `[first, first)`.
| `match_not_eol`  | The last character in the sequence `[first, last)` shall be treated as though it is not at the end of a line, so the character `$` in the regular expression shall not match `[last, last)`.
| `match_not_bow`  | The expression `\b` shall not match the sub-sequence `[first, first)`.
| `match_not_eow`  | The expression `\b` shall not match the sub-sequence `[last, last)`.
| `match_any`      | If more than one match is possible then any match is an acceptable result.
| `match_not_null` | The expression shall not match an empty sequence.
| `match_continuous` | The expression shall only match a sub-sequence that begins at `first`.
| `match_prev_avail` | `--first` is a valid iterator position. When this flag is set the flags `match_not_bol` and `match_not_bow` shall be ignored by the regular expression algorithms (30.10) and iterators (30.11).
| `format_default` | When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the ECMAScript replace function in ECMA-262, part 15.5.4.11 String.prototype.replace. In addition, during search and replace operations all non-overlapping occurrences of the regular expression shall be located and replaced, and sections of the input that did not match the expression shall be copied unchanged to the output string.
| `format_sed`     | When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the sed utility in POSIX.
| `format_no_copy` | During a search and replace operation, sections of the character container sequence being searched that do not match the regular expression shall not be copied to the output string.
| `format_first_only` | When specified during a search and replace operation, only the first occurrence of the regular expression shall be replaced.

30.4.4 Implementation-defined error_type

```cpp
namespace std::regex_constants {
    using error_type = T3;
}
```
The type `error_type` is an implementation-defined enumerated type (16.3.3.3.3). Values of type `error_type` represent the error conditions described in Table 138:

<table>
<thead>
<tr>
<th>Value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>error_collate</code></td>
<td>The expression contained an invalid collating element name.</td>
</tr>
<tr>
<td><code>error_ctype</code></td>
<td>The expression contained an invalid character class name.</td>
</tr>
<tr>
<td><code>error_escape</code></td>
<td>The expression contained an invalid escaped character, or a trailing escape.</td>
</tr>
<tr>
<td><code>error_backref</code></td>
<td>The expression contained an invalid back reference.</td>
</tr>
<tr>
<td><code>error_brack</code></td>
<td>The expression contained mismatched <code>[</code> and <code>]</code>.</td>
</tr>
<tr>
<td><code>error_paren</code></td>
<td>The expression contained mismatched <code>(</code> and <code>)</code>.</td>
</tr>
<tr>
<td><code>error_brace</code></td>
<td>The expression contained mismatched <code>{</code> and <code>}</code>.</td>
</tr>
<tr>
<td><code>error_badbrace</code></td>
<td>The expression contained an invalid range in a <code>{}</code> expression.</td>
</tr>
<tr>
<td><code>error_range</code></td>
<td>The expression contained an invalid character range, such as <code>[b-a]</code> in most encodings.</td>
</tr>
<tr>
<td><code>error_space</code></td>
<td>There was insufficient memory to convert the expression into a finite state machine.</td>
</tr>
<tr>
<td><code>error_badrepeat</code></td>
<td>One of <code>*</code>?{` was not preceded by a valid regular expression.</td>
</tr>
<tr>
<td><code>error_complexity</code></td>
<td>The complexity of an attempted match against a regular expression exceeded a pre-set level.</td>
</tr>
<tr>
<td><code>error_stack</code></td>
<td>There was insufficient memory to determine whether the regular expression could match the specified character sequence.</td>
</tr>
</tbody>
</table>

30.5 Class `regex_error` [re.badexp]

```cpp
class regex_error : public runtime_error {
public:
    explicit regex_error(regex_constants::error_type ecode);
    regex_constants::error_type code() const;
};
```

1 The class `regex_error` defines the type of objects thrown as exceptions to report errors from the regular expression library.

```cpp
regex_error(regex_constants::error_type ecode);
```

2 Postconditions: `ecode == code()`.

```cpp
regex_constants::error_type code() const;
```

3 Returns: The error code that was passed to the constructor.

30.6 Class template `regex_traits` [re.traits]

```cpp
namespace std {
    template<class charT>
    struct regex_traits {
        using char_type = charT;
        using string_type = basic_string<char_type>;
    }
}
```
using locale_type = locale;
using char_class_type = bitmask_type;

regex_traits();
static size_t length(const char_type* p);
charT translate(charT c) const;
charT translate_nocase(charT c) const;
template<class ForwardIterator>
  string_type transform(ForwardIterator first, ForwardIterator last) const;
template<class ForwardIterator>
  string_type transform_primary(ForwardIterator first, ForwardIterator last) const;
template<class ForwardIterator>
  string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;
template<class ForwardIterator>
  char_class_type lookup_classname(ForwardIterator first, ForwardIterator last, bool icase = false) const;
bool isctype(charT c, char_class_type f) const;
int value(charT ch, int radix) const;
locale_type imbue(locale_type l);
locale_type getloc() const;
}

1 The specializations regex_traits<char> and regex_traits<wchar_t> meet the requirements for a regular expression traits class (30.2).

using char_class_type = bitmask_type;

2 The type char_class_type is used to represent a character classification and is capable of holding an implementation specific set returned by lookup_classname.

static size_t length(const char_type* p);

3 Returns: char_traits<charT>::length(p).

charT translate(charT c) const;

4 Returns: c.

charT translate_nocase(charT c) const;

5 Returns: use_facet<ctype<charT>>(getloc()).tolower(c).

template<class ForwardIterator>
  string_type transform(ForwardIterator first, ForwardIterator last) const;

6 Effects: As if by:
  string_type str(first, last);
  return use_facet<collate<charT>>(getloc()).transform(str.data(), str.data() + str.length());

template<class ForwardIterator>
  string_type transform_primary(ForwardIterator first, ForwardIterator last) const;

7 Effects: If
typeid(use_facet<collate<charT>>) == typeid(collate_byname<charT>)
and the form of the sort key returned by collate_byname<charT>::transform(first, last) is known and can be converted into a primary sort key then returns that key, otherwise returns an empty string.

template<class ForwardIterator>
  string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;

8 Returns: A sequence of one or more characters that represents the collating element consisting of the character sequence designated by the iterator range [first, last). Returns an empty string if the character sequence is not a valid collating element.

§ 30.6 1515
template<class ForwardIterator>
char_class_type lookup_classname(ForwardIterator first, ForwardIterator last, bool icase = false) const;

Returns: An unspecified value that represents the character classification named by the character sequence designated by the iterator range [first, last). If the parameter icase is true then the returned mask identifies the character classification without regard to the case of the characters being matched, otherwise it does honor the case of the characters being matched.\textsuperscript{327} The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns char_class_type().

Remarks: For regex_traits<char>, at least the narrow character names in Table 139 shall be recognized. For regex_traits<wchar_t>, at least the wide character names in Table 139 shall be recognized.

bool isctype(charT c, char_class_type f) const;

Effects: Determines if the character c is a member of the character classification represented by f.

Returns: Given the following function declaration:

```cpp
// for exposition only
template<class C>
ctype_base::mask convert(typename regex_traits<C>::char_class_type f); //...\n```

that returns a value in which each ctype_base::mask value corresponding to a value in f named in Table 139 is set, then the result is determined as if by:

```cpp
cctype_base::mask m = convert<charT>(f);
const ctype<charT>& ct = use_facet<ctype<charT>>(getloc());
if (ct.is(m, c)) { return true; }
else if (c == ct.widen('_')) {
    charT w[1] = { ct.widen('w') };
    char_class_type x = lookup_classname(w, w+1);
    return (f&x) == x;
} else {
    return false;
}
```

[Example 1:]

```cpp
regex_traits<char> t;
string d("d");
string u("upper");
regex_traits<char>::char_class_type f;
f |= t.lookup_classname(d.begin(), d.end());
f |= t.lookup_classname(u.begin(), u.end());
cctype_base::mask m = convert<char>(f); // m == ctype_base::digit|ctype_base::upper
```

— end example]

[Example 2:]

```cpp
regex_traits<char> t;
string w("w");
regex_traits<char>::char_class_type f;
f = t.lookup_classname(w.begin(), w.end());
t.isctype(\'A\', f); // returns true
t.isctype(\'a\', f); // returns true
t.isctype(\'\', f); // returns false
```

— end example]

int value(charT ch, int radix) const;

Preconditions: The value of radix is 8, 10, or 16.

Returns: The value represented by the digit ch in base radix if the character ch is a valid digit in base radix; otherwise returns -1.

\textsuperscript{327} For example, if the parameter icase is true then \([[:lower:]]\) is the same as \([[:alpha:]]\).
locale_type imbue(locale_type loc);

Effects: Imbues this with a copy of the locale loc.

[Note 1: Calling imbue with a different locale than the one currently in use invalidates all cached data held by *this. —end note]

Postconditions: getloc() == loc.

Returns: If no locale has been previously imbued then a copy of the global locale in effect at the time of construction of *this, otherwise a copy of the last argument passed to imbue.

locale_type getloc() const;

Returns: If no locale has been imbued then a copy of the global locale in effect at the time of construction of *this, otherwise a copy of the last argument passed to imbue.

Table 139: Character class names and corresponding ctype masks

<table>
<thead>
<tr>
<th>Narrow character name</th>
<th>Wide character name</th>
<th>Corresponding ctype_base::mask value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;alnum&quot;</td>
<td>L&quot;alnum&quot;</td>
<td>ctype_base::alnum</td>
</tr>
<tr>
<td>&quot;alpha&quot;</td>
<td>L&quot;alpha&quot;</td>
<td>ctype_base::alpha</td>
</tr>
<tr>
<td>&quot;blank&quot;</td>
<td>L&quot;blank&quot;</td>
<td>ctype_base::blank</td>
</tr>
<tr>
<td>&quot;cntrl&quot;</td>
<td>L&quot;cntrl&quot;</td>
<td>ctype_base::cntrl</td>
</tr>
<tr>
<td>&quot;digit&quot;</td>
<td>L&quot;digit&quot;</td>
<td>ctype_base::digit</td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>L&quot;d&quot;</td>
<td>ctype_base::digit</td>
</tr>
<tr>
<td>&quot;graph&quot;</td>
<td>L&quot;graph&quot;</td>
<td>ctype_base::graph</td>
</tr>
<tr>
<td>&quot;lower&quot;</td>
<td>L&quot;lower&quot;</td>
<td>ctype_base::lower</td>
</tr>
<tr>
<td>&quot;print&quot;</td>
<td>L&quot;print&quot;</td>
<td>ctype_base::print</td>
</tr>
<tr>
<td>&quot;punct&quot;</td>
<td>L&quot;punct&quot;</td>
<td>ctype_base::punct</td>
</tr>
<tr>
<td>&quot;space&quot;</td>
<td>L&quot;space&quot;</td>
<td>ctype_base::space</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>L&quot;s&quot;</td>
<td>ctype_base::space</td>
</tr>
<tr>
<td>&quot;upper&quot;</td>
<td>L&quot;upper&quot;</td>
<td>ctype_base::upper</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>L&quot;w&quot;</td>
<td>ctype_base::alnum</td>
</tr>
<tr>
<td>&quot;xdigit&quot;</td>
<td>L&quot;xdigit&quot;</td>
<td>ctype_base::xdigit</td>
</tr>
</tbody>
</table>

30.7 Class template basic_regex

30.7.1 General

For a char-like type charT, specializations of class template basic_regex represent regular expressions constructed from character sequences of charT characters. In the rest of 30.7, charT denotes a given char-like type. Storage for a regular expression is allocated and freed as necessary by the member functions of class basic_regex.

Objects of type specialization of basic_regex are responsible for converting the sequence of charT objects to an internal representation. It is not specified what form this representation takes, nor how it is accessed by algorithms that operate on regular expressions.

[Note 1: Implementations will typically declare some function templates as friends of basic_regex to achieve this. —end note]

The functions described in 30.7 report errors by throwing exceptions of type regex_error.
// 30.4.2, constants
static constexpr flag_type icase = regex_constants::icase;
static constexpr flag_type nosubs = regex_constants::nosubs;
static constexpr flag_type optimize = regex_constants::optimize;
static constexpr flag_type collate = regex_constants::collate;
static constexpr flag_type ECMAScript = regex_constants::ECMAScript;
static constexpr flag_type basic = regex_constants::basic;
static constexpr flag_type extended = regex_constants::extended;
static constexpr flag_type awk = regex_constants::awk;
static constexpr flag_type grep = regex_constants::grep;
static constexpr flag_type egrep = regex_constants::egrep;
static constexpr flag_type multiline = regex_constants::multiline;

// 30.7.2, construct/copy/destroy
basic_regex();
explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
basic_regex(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);
basic_regex(const basic_regex&);
basic_regex(basic_regex&&) noexcept;
template<class ST, class SA>
  explicit basic_regex(const basic_string<charT, ST, SA>& s, flag_type f = regex_constants::ECMAScript);
template<class ForwardIterator>
  basic_regex(ForwardIterator first, ForwardIterator last, flag_type f = regex_constants::ECMAScript);
basic_regex(initializer_list<charT> il, flag_type f = regex_constants::ECMAScript);

~basic_regex();

// 30.7.3, assign
basic_regex& operator=(const basic_regex& e);
basic_regex& operator=(basic_regex&& e) noexcept;
basic_regex& operator=(const charT* p);
basic_regex& operator=(initializer_list<charT> il);
template<class ST, class SA>
  basic_regex& operator=(const basic_string<charT, ST, SA>& s);

basic_regex& assign(const basic_regex& e);
basic_regex& assign(basic_regex&& e) noexcept;
basic_regex& assign(const charT* p, flag_type f = regex_constants::ECMAScript);
basic_regex& assign(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);
template<class ST, class SA>
  basic_regex& assign(const basic_string<charT, ST, SA>& s, flag_type f = regex_constants::ECMAScript);
template<class InputIterator>
  basic_regex& assign(InputIterator first, InputIterator last, flag_type f = regex_constants::ECMAScript);

// 30.7.4, const operations
unsigned mark_count() const;
flag_type flags() const;

// 30.7.5, locale
locale_type imbue(locale_type loc);
locale_type getloc() const;

// 30.7.6, swap
void swap(basic_regex&);
template<class ForwardIterator>
    basic_regex(ForwardIterator, ForwardIterator,
                regex_constants::syntax_option_type = regex_constants::ECMAScript)
    -> basic_regex<typename iterator_traits<ForwardIterator>::value_type>;

30.7.2 Constructors

basic_regex();
Postconditions: *this does not match any character sequence.

explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
Preconditions: [p, p + char_traits<charT>::length(p)) is a valid range.
Effects: The object’s internal finite state machine is constructed from the regular expression contained in
the sequence of characters [p, p + char_traits<charT>::length(p)), and interpreted according to the flags f.
Postconditions: flags() returns f. mark_count() returns the number of marked sub-expressions
within the expression.
Throws: regex_error if [p, p + char_traits<charT>::length(p)) is not a valid regular expression.

basic_regex(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);
Preconditions: [p, p + len) is a valid range.
Effects: The object’s internal finite state machine is constructed from the regular expression contained in
the sequence of characters [p, p + len), and interpreted according the flags specified in f.
Postconditions: flags() returns f. mark_count() returns the number of marked sub-expressions
within the expression.
Throws: regex_error if [p, p + len) is not a valid regular expression.

basic_regex(const basic_regex& e);
Postconditions: flags() and mark_count() return e.flags() and e.mark_count(), respectively.

basic_regex(basic_regex&& e) noexcept;
Postconditions: flags() and mark_count() return the values that e.flags() and e.mark_count(),
respectively, had before construction.

template<class ST, class SA>
    explicit basic_regex(const basic_string<charT, ST, SA>& s,
                        flag_type f = regex_constants::ECMAScript);
Effects: The object’s internal finite state machine is constructed from the regular expression contained
in the string s, and interpreted according to the flags specified in f.
Postconditions: flags() returns f. mark_count() returns the number of marked sub-expressions
within the expression.
Throws: regex_error if s is not a valid regular expression.

template<class ForwardIterator>
    basic_regex(ForwardIterator first, ForwardIterator last,
                flag_type f = regex_constants::ECMAScript);
Effects: The object’s internal finite state machine is constructed from the regular expression contained in
the sequence of characters [first, last), and interpreted according to the flags specified in f.
Postconditions: flags() returns f. mark_count() returns the number of marked sub-expressions
within the expression.
Throws: regex_error if the sequence [first, last) is not a valid regular expression.

basic_regex(initializer_list<charT> il, flag_type f = regex_constants::ECMAScript);
Effects: Same as basic_regex(il.begin(), il.end(), f).
30.7.3 Assignment

basic_regex& operator=(const basic_regex& e);

Postconditions: flags() and mark_count() return e.flags() and e.mark_count(), respectively.

basic_regex& operator=(basic_regex&& e) noexcept;

Postconditions: flags() and mark_count() return the values that e.flags() and e.mark_count(), respectively, had before assignment. e is in a valid state with unspecified value.

basic_regex& operator=(const charT* p);

Effects: Equivalent to: return assign(p);

basic_regex& operator=(initializer_list<charT> il);

Effects: Equivalent to: return assign(il.begin(), il.end());

template<class ST, class SA>
basic_regex& operator=(const basic_string<charT, ST, SA>& s);

Effects: Equivalent to: return assign(s);

basic_regex& assign(const basic_regex& e);

Effects: Equivalent to: return *this = e;

basic_regex& assign(basic_regex&& e) noexcept;

Effects: Equivalent to: return *this = std::move(e);

basic_regex& assign(const charT* p, flag_type f = regex_constants::ECMAScript);

Effects: Equivalent to: return assign(string_type(p), f);

basic_regex& assign(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);

Effects: Equivalent to: return assign(string_type(p, len), f);

template<class ST, class SA>
basic_regex& assign(const basic_string<charT, ST, SA>& s, flag_type f = regex_constants::ECMAScript);

Returns: *this.

Effects: Assigns the regular expression contained in the string s, interpreted according the flags specified in f. If an exception is thrown, *this is unchanged.

Postconditions: If no exception is thrown, flags() returns f and mark_count() returns the number of marked sub-expressions within the expression.

Throws: regex_error if s is not a valid regular expression.

template<class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last, flag_type f = regex_constants::ECMAScript);

Effects: Equivalent to: return assign(string_type(first, last), f);

basic_regex& assign(initializer_list<charT> il, flag_type f = regex_constants::ECMAScript);

Effects: Equivalent to: return assign(il.begin(), il.end(), f);

30.7.4 Constant operations

unsigned mark_count() const;

Effects: Returns the number of marked sub-expressions within the regular expression.

flag_type flags() const;

Effects: Returns a copy of the regular expression syntax flags that were passed to the object’s constructor or to the last call to assign.
30.7.5 Locale

locale_type imbue(locale_type loc);

*Effects*: Returns the result of traits_inst.imbue(loc) where traits_inst is a (default-initialized) instance of the template type argument traits stored within the object. After a call to imbue the basic_regex object does not match any character sequence.

locale_type getloc() const;

*Effects*: Returns the result of traits_inst.getloc() where traits_inst is a (default-initialized) instance of the template parameter traits stored within the object.

30.7.6 Swap

void swap(basic_regex& e);

*Effects*: Swaps the contents of the two regular expressions.

*Postconditions*: *this contains the regular expression that was in e, e contains the regular expression that was in *this.

*Complexity*: Constant time.

30.7.7 Non-member functions

template<class charT, class traits>

void swap(basic_regex<charT, traits>& lhs, basic_regex<charT, traits>& rhs);

*Effects*: Calls lhs.swap(rhs).

30.8 Class template sub_match

30.8.1 General

namespace std {

    template<class BidirectionalIterator>

class sub_match : public pair<BidirectionalIterator, BidirectionalIterator> {

public:

    using value_type = typename iterator_traits<BidirectionalIterator>::value_type;

    using difference_type = typename iterator_traits<BidirectionalIterator>::difference_type;

    using iterator = BidirectionalIterator;

    using string_type = basic_string<value_type>;

    bool matched;

    constexpr sub_match();

    difference_type length() const;

    operator string_type() const;

    string_type str() const;

    int compare(const sub_match& s) const;

    int compare(const string_type& s) const;

    int compare(const value_type* s) const;

};

}

30.8.2 Members

constexpr sub_match();

*Effects*: Value-initializes the pair base class subobject and the member matched.

difference_type length() const;

*Returns*: matched ? distance(first, second) : 0.
operator string_type() const;

    Returns: matched ? string_type(first, second) : string_type().

string_type str() const;

    Returns: matched ? string_type(first, second) : string_type().

int compare(const sub_match& s) const;

    Returns: str().compare(s.str()).

int compare(const string_type& s) const;

    Returns: str().compare(s).

int compare(const value_type* s) const;

    Returns: str().compare(s).

30.8.3 Non-member operators [re.submatch.op]

Let SM-CAT(I) be

    compare_three_way_result_t<basic_string<typename iterator_traits<I>::value_type>>

template<class BiIter>
    bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

    Returns: lhs.compare(rhs) == 0.

template<class BiIter>
    auto operator<=>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

    Returns: static_cast<SM-CAT(BiIter)>(lhs.compare(rhs) <=> 0).

template<class BiIter, class ST, class SA>
    bool operator==(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

    Returns: lhs.compare(typename sub_match<BiIter>::string_type(rhs.data(), rhs.size())) == 0.

template<class BiIter, class ST, class SA>
    auto operator<=>(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

    Returns: static_cast<SM-CAT(BiIter)>(lhs.compare( typename sub_match<BiIter>::string_type(rhs.data(), rhs.size())) <=> 0)

template<class BiIter>
    bool operator==(const sub_match<BiIter>& lhs, const typename iterator_traits<BiIter>::value_type* rhs);

    Returns: lhs.compare(rhs) == 0.

template<class BiIter>
    auto operator<=>(const sub_match<BiIter>& lhs, const typename iterator_traits<BiIter>::value_type* rhs);

    Returns: static_cast<SM-CAT(BiIter)>(lhs.compare(rhs) <=> 0).

template<class BiIter>
    bool operator==(const sub_match<BiIter>& lhs, const typename iterator_traits<BiIter>::value_type& rhs);

    Returns: lhs.compare(typename sub_match<BiIter>::string_type(1, rhs)) == 0.
template<class BiIter>
auto operator<=>(const sub_match<BiIter>& lhs,
    const typename iterator_traits<BiIter>::value_type& rhs);

9
Returns:
static_cast<SM-CAT(BiIter)>(lhs.compare(
    typename sub_match<BiIter>::string_type(1, rhs))
    <=> 0
)

template<class charT, class ST, class BiIter>
basic_ostream<charT, ST>&
operator<<(basic_ostream<charT, ST>&& os, const sub_match<BiIter>& m);

10
Returns: os << m.str().

30.9 Class template match_results
30.9.1 General
1
Class template match_results denotes a collection of character sequences representing the result of a regular
expression match. Storage for the collection is allocated and freed as necessary by the member functions of
class template match_results.

2
The class template match_results meets the requirements of an allocator-aware container and of a sequence
container (22.2.1, 22.2.3) except that only copy assignment, move assignment, and operations defined for
const-qualified sequence containers are supported and that the semantics of the comparison operator functions
are different from those required for a container.

3
A default-constructed match_results object has no fully established result state. A match result is ready
when, as a consequence of a completed regular expression match modifying such an object, its result state
becomes fully established. The effects of calling most member functions from a match_results object that
is not ready are undefined.

4
The sub_match object stored at index 0 represents sub-expression 0, i.e., the whole match. In this case the
sub_match member matched is always true. The sub_match object stored at index n denotes what matched
the marked sub-expression n within the matched expression. If the sub-expression n participated in a regular
expression match then the sub_match member matched evaluates to true, and members first and second
denote the range of characters [first, second) which formed that match. Otherwise matched is false,
and members first and second point to the end of the sequence that was searched.

[Note 1: The sub_match objects representing different sub-expressions that did not participate in a regular expression
match need not be distinct. — end note]

namespace std {
    template<class BidirectionalIterator,
        class Allocator = allocator<sub_match<BidirectionalIterator>>>
    class match_results {
        public:
            using value_type = sub_match<BidirectionalIterator>;
            using const_reference = const value_type&;
            using reference = value_type&;
            using const_iterator = implementation-defined;
            using iterator = const_iterator;
            using difference_type =
                typename iterator_traits<BidirectionalIterator>::difference_type;
            using size_type = typename allocator_traits<Allocator>::size_type;
            using allocator_type = Allocator;
            using char_type =
                typename iterator_traits<BidirectionalIterator>::value_type;
            using string_type = basic_string<char_type>;

        // 30.9.2, construct/copy/destroy
        match_results() : match_results(Allocator()) {}
        explicit match_results(const Allocator&);
        match_results(const match_results& m);
        match_results(match_results&& m) noexcept;

        § 30.9.1 1523
30.9.2 Constructors

```cpp
explicit match_results(const Allocator& a);

Postconditions: ready() returns false. size() returns 0.

match_results& operator=(const match_results& m);
match_results& operator=(match_results&& m);
~match_results();
```

30.9.3 state

```cpp
bool ready() const;
```

30.9.4 size

```cpp
size_type size() const;
size_type max_size() const;
[[nodiscard]] bool empty() const;
```

30.9.5 element access

```cpp
difference_type length(size_type sub = 0) const;
difference_type position(size_type sub = 0) const;
string_type str(size_type sub = 0) const;
const_reference operator[](size_type n) const;
```

```cpp
const_reference prefix() const;
const_reference suffix() const;
const_iterator begin() const;
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;
```

30.9.6 format

```cpp
template<class OutputIter>
OutputIter
format(OutputIter out,
      const char_type* fmt_first, const char_type* fmt_last,
      regex_constants::match_flag_type flags = regex_constants::format_default) const;
template<class OutputIter, class ST, class SA>
OutputIter
format(OutputIter out,
      const basic_string<char_type, ST, SA>& fmt,
      regex_constants::match_flag_type flags = regex_constants::format_default) const;
template<class ST, class SA>
basic_string<char_type, ST, SA>
format(const basic_string<char_type, ST, SA>& fmt,
       regex_constants::match_flag_type flags = regex_constants::format_default) const;
string_type
format(const char_type* fmt,
       regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

30.9.7 allocator

```cpp
allocator_type get_allocator() const;
```

30.9.8 swap

```cpp
void swap(match_results& that);
```

match_results& operator=(match_results&& m);

Postconditions: As specified in Table 140.

Table 140: match_results assignment operator effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready()</td>
<td>m.ready()</td>
</tr>
<tr>
<td>size()</td>
<td>m.size()</td>
</tr>
<tr>
<td>str(n)</td>
<td>m.str(n) for all integers n &lt; m.size()</td>
</tr>
<tr>
<td>prefix()</td>
<td>m.prefix()</td>
</tr>
<tr>
<td>suffix()</td>
<td>m.suffix()</td>
</tr>
<tr>
<td>(*this)[n]</td>
<td>m[n] for all integers n &lt; m.size()</td>
</tr>
<tr>
<td>length(n)</td>
<td>m.length(n) for all integers n &lt; m.size()</td>
</tr>
<tr>
<td>position(n)</td>
<td>m.position(n) for all integers n &lt; m.size()</td>
</tr>
</tbody>
</table>

30.9.3 State

bool ready() const;

Returns: true if *this has a fully established result state, otherwise false.

30.9.4 Size

size_type size() const;

Returns: One plus the number of marked sub-expressions in the regular expression that was matched if *this represents the result of a successful match. Otherwise returns 0.

[Note 1: The state of a match_results object can be modified only by passing that object to regex_match or regex_search. Subclauses 30.10.2 and 30.10.3 specify the effects of those algorithms on their match_results arguments. — end note]

size_type max_size() const;

Returns: The maximum number of sub_match elements that can be stored in *this.

[nodiscard] bool empty() const;

Returns: size() == 0.

30.9.5 Element access

difference_type length(size_type sub = 0) const;

Preconditions: ready() == true.

Returns: (*this)[sub].length().

difference_type position(size_type sub = 0) const;

Preconditions: ready() == true.

Returns: The distance from the start of the target sequence to (*this)[sub].first.

string_type str(size_type sub = 0) const;

Preconditions: ready() == true.

Returns: string_type(*this)[sub]).

cost_reference operator[](size_type n) const;

Preconditions: ready() == true.

Returns: A reference to the sub_match object representing the character sequence that matched marked sub-expression n. If n == 0 then returns a reference to a sub_match object representing the character sequence that matched the whole regular expression. If n >= size() then returns a sub_match object representing an unmatched sub-expression.
const_reference prefix() const;

   *Preconditions:* ready() == true.

   *Returns:* A reference to the sub_match object representing the character sequence from the start of the string being matched/searched to the start of the match found.

const_reference suffix() const;

   *Preconditions:* ready() == true.

   *Returns:* A reference to the sub_match object representing the character sequence from the end of the match found to the end of the string being matched/searched.

const_iterator begin() const;
const_iterator cbegin() const;

   *Returns:* A starting iterator that enumerates over all the sub-expressions stored in *this.

const_iterator end() const;
const_iterator cend() const;

   *Returns:* A terminating iterator that enumerates over all the sub-expressions stored in *this.

### 30.9.6 Formatting

```cpp
template<class OutputIter>
OutputIter format(
    OutputIter out,
    const char_type* fmt_first, const char_type* fmt_last,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

   *Preconditions:* ready() == true and OutputIter meets the requirements for a Cpp17OutputIterator (23.3.5.4).

   *Effects:* Copies the character sequence [fmt_first, fmt_last) to OutputIter out. Replaces each format specifier or escape sequence in the copied range with either the character(s) it represents or the sequence of characters within *this to which it refers. The bitmasks specified in flags determine which format specifiers and escape sequences are recognized.

   *Returns:* out.

```cpp
template<class OutputIter, class ST, class SA>
OutputIter format(
    OutputIter out,
    const basic_string<char_type, ST, SA>& fmt,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

   *Effects:* Equivalent to:

   ```cpp
   return format(out, fmt.data(), fmt.data() + fmt.size(), flags);
   ```

```cpp
template<class ST, class SA>
basic_string<char_type, ST, SA> format(
    const basic_string<char_type, ST, SA>& fmt,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

   *Preconditions:* ready() == true.

   *Effects:* Constructs an empty string result of type basic_string<char_type, ST, SA> and calls: `format(back_inserter(result), fmt, flags);`

   *Returns:* result.

```cpp
string_type format(
    const char_type* fmt,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

   *Preconditions:* ready() == true.

   *Effects:* Constructs an empty string result of type string_type and calls: `format(back_inserter(result), fmt, fmt + char_traits<char_type>::length(fmt), flags);`

§ 30.9.6 1526
Returns: result.

30.9.7 Allocator

```cpp
allocator_type get_allocator() const;
```

Returns: A copy of the Allocator that was passed to the object’s constructor or, if that allocator has been replaced, a copy of the most recent replacement.

30.9.8 Swap

```cpp
void swap(match_results& that);
```

Effects: Swaps the contents of the two sequences.

Postconditions: `*this` contains the sequence of matched sub-expressions that were in `that`, `that` contains the sequence of matched sub-expressions that were in `*this`.

Complexity: Constant time.

```cpp
template<class BidirectionalIterator, class Allocator>
test swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);
```

Effects: As if by `m1.swap(m2)`.

30.9.9 Non-member functions

```cpp
test operator==(const match_results<BidirectionalIterator, Allocator>& m1,
              const match_results<BidirectionalIterator, Allocator>& m2);
```

Returns: `true` if neither match result is ready, `false` if one match result is ready and the other is not. If both match results are ready, returns `true` only if:

1. `m1.empty() && m2.empty()`, or
2. `!m1.empty() && !m2.empty()`, and the following conditions are satisfied:
   1. `m1.prefix() == m2.prefix()`,
   2. `m1.size() == m2.size() && equal(m1.begin(), m1.end(), m2.begin())`, and
   3. `m1.suffix() == m2.suffix()`.

[Note 1: The algorithm `equal` is defined in Clause 25. — end note]

30.10 Regular expression algorithms

30.10.1 Exceptions

The algorithms described in subclause 30.10 may throw an exception of type `regex_error`. If such an exception `e` is thrown, `e.code()` shall return either `regex_constants::error_complexity` or `regex_constants::error_stack`.

30.10.2 `regex_match`

```cpp
test regex_match(BidirectionalIterator first, BidirectionalIterator last,
                  match_results<BidirectionalIterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
```

Preconditions: `BidirectionalIterator` meets the `Cpp17BidirectionalIterator` requirements (23.3.5.6). Effects: Determines whether there is a match between the regular expression `e`, and all of the character sequence `[first, last)`. The parameter `flags` is used to control how the expression is matched against the character sequence. When determining if there is a match, only potential matches that match the entire character sequence are considered. Returns `true` if such a match exists, `false` otherwise.

[Example 1:]
```cpp```
std::regex re("Get|GetValue");
Postconditions: `m.ready()` == `true` in all cases. If the function returns `false`, then the effect on parameter `m` is unspecified except that `m.size()` returns `0` and `m.empty()` returns `true`. Otherwise the effects on parameter `m` are given in Table 141.

Table 141: Effects of `regex_match` algorithm

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m.size()</code></td>
<td><code>1 + e.mark_count()</code></td>
</tr>
<tr>
<td><code>m.empty()</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>m.prefix().first</code></td>
<td><code>first</code></td>
</tr>
<tr>
<td><code>m.prefix().second</code></td>
<td><code>first</code></td>
</tr>
<tr>
<td><code>m.prefix().matched</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>m.suffix().first</code></td>
<td><code>last</code></td>
</tr>
<tr>
<td><code>m.suffix().second</code></td>
<td><code>last</code></td>
</tr>
<tr>
<td><code>m.suffix().matched</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>m[0].first</code></td>
<td><code>first</code></td>
</tr>
<tr>
<td><code>m[0].second</code></td>
<td><code>last</code></td>
</tr>
<tr>
<td><code>m[0].matched</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>m[n].first</code></td>
<td>For all integers <code>0 &lt; n &lt; m.size()</code>, the start of the sequence that matched sub-expression <code>n</code>. Alternatively, if sub-expression <code>n</code> did not participate in the match, then last.</td>
</tr>
<tr>
<td><code>m[n].second</code></td>
<td>For all integers <code>0 &lt; n &lt; m.size()</code>, the end of the sequence that matched sub-expression <code>n</code>. Alternatively, if sub-expression <code>n</code> did not participate in the match, then last.</td>
</tr>
<tr>
<td><code>m[n].matched</code></td>
<td>For all integers <code>0 &lt; n &lt; m.size()</code>, <code>true</code> if sub-expression <code>n</code> participated in the match, <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>

§ 30.10.2 1528
template<class charT, class traits>
bool regex_match(const charT* str,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_match(str, str + char_traits<charT>::length(str), e, flags)

template<class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_match(s.begin(), s.end(), e, flags).

30.10.3 regex_search

template<class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
    match_results<BidirectionalIterator, Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

Preconditions: BidirectionalIterator meets the C++17BidirectionalIterator requirements (23.3.5.6).

Effects: Determines whether there is some sub-sequence within [first, last) that matches the regular expression e. The parameter flags is used to control how the expression is matched against the character sequence. Returns true if such a sequence exists, false otherwise.

Postconditions: m.ready() == true in all cases. If the function returns false, then the effect on parameter m is unspecified except that m.size() returns 0 and m.empty() returns true. Otherwise the effects on parameter m are given in Table 142.

Table 142: Effects of regex_search algorithm

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td>1 + e.mark_count()</td>
</tr>
<tr>
<td>m.empty()</td>
<td>false</td>
</tr>
<tr>
<td>m.prefix().first</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().second</td>
<td>m[0].first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>m.prefix().first != m.prefix().second</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>m[0].second</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>m.suffix().first != m.suffix().second</td>
</tr>
<tr>
<td>m[0].first</td>
<td>The start of the sequence of characters that matched the regular expression</td>
</tr>
<tr>
<td>m[0].second</td>
<td>The end of the sequence of characters that matched the regular expression</td>
</tr>
<tr>
<td>m[0].matched</td>
<td>true</td>
</tr>
<tr>
<td>m[n].first</td>
<td>For all integers 0 &lt; n &lt; m.size(), the start of the sequence that matched sub-expression n. Alternatively, if sub-expression n did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].second</td>
<td>For all integers 0 &lt; n &lt; m.size(), the end of the sequence that matched sub-expression n. Alternatively, if sub-expression n did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].matched</td>
<td>For all integers 0 &lt; n &lt; m.size(), true if sub-expression n participated in the match, false otherwise.</td>
</tr>
</tbody>
</table>

template<class charT, class Allocator, class traits>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
    const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_search(str, str + char_traits<charT>::length(str), m, e, flags).

template<class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
match_results<typename basic_string<charT, ST, SA>::const_iterator,
Allocator>& m,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_search(s.begin(), s.end(), m, e, flags).

template<class BidirectionalIterator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags = regex_constants::match_default);

Effects: Behaves “as if” by constructing an object what of type match_results<BidirectionalIterator> and returning regex_search(first, last, what, e, flags).

template<class charT, class traits>
bool regex_search(const charT* str,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_search(str, str + char_traits<charT>::length(str), e, flags).

template<class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_search(s.begin(), s.end(), e, flags).

30.10.4 regex_replace

template<class OutputIterator, class BidirectionalIterator,
class traits, class charT, class ST, class SA>
OutputIterator
regex_replace(OutputIterator out,
BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
const basic_string<charT, ST, SA>& fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);

template<class OutputIterator, class BidirectionalIterator, class traits, class charT>
OutputIterator
regex_replace(OutputIterator out,
BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);

Effects: Constructs a regex_iterator object i as if by
regex_iterator<BidirectionalIterator, charT, traits> i(first, last, e, flags)
and uses i to enumerate through all of the matches m of type match_results<BidirectionalIterator>
that occur within the sequence [first, last). If no such matches are found and !(flags & regex_constants::format_no_copy), then calls
out = copy(first, last, out)
If any matches are found then, for each such match:

(1.1) — If !(flags & regex_constants::format_no_copy), calls
out = copy(m.prefix().first, m.prefix().second, out)
(1.2) — Then calls
out = m.format(out, fmt, flags)
for the first form of the function and
\[
\text{out} = \text{m.format(out, fmt, fmt + char_traits<charT>::length(fmt), flags)}
\]
for the second.

Finally, if such a match is found and !(flags & regex_constants::format_no_copy), calls
\[
\text{out} = \text{copy(last_m.suffix().first, last_m.suffix().second, out)}
\]
where last_m is a copy of the last match found. If flags & regex_constants::format_first_only is nonzero, then only the first match found is replaced.

Returns: out.

```
template<class traits, class charT, class ST, class SA, class FST, class FSA>
basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
const basic_string<charT, FST, FSA>& fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);
```

Effects: Constructs an empty string result of type basic_string<charT, ST, SA> and calls:
\[
\text{regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags)};
\]

Returns: result.

```
template<class traits, class charT, class ST, class SA>
basic_string<charT>
regex_replace(const charT* s,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);
```

Effects: Constructs an empty string result of type basic_string<charT> and calls:
\[
\text{regex_replace(back_inserter(result), s, s + char_traits<charT>::length(s), e, fmt, flags)};
\]

Returns: result.

### 30.11 Regular expression iterators

#### 30.11.1 Class template regex_iterator

The class template regex_iterator is an iterator adaptor. It represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. A regex_iterator uses regex_search to find successive regular expression matches within the sequence from which it was constructed. After the iterator is constructed, and every time operator++ is used, the iterator finds and stores a value of match_results<BidirectionalIterator>. If the end of the sequence is reached (regex_search returns false), the iterator becomes equal to the end-of-sequence iterator value. The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-sequence iterator is not defined. For any other iterator value a const match_results<BidirectionalIterator>& is returned. The result of operator-> on an end-of-sequence iterator is not defined. For any other iterator value a const match_results<BidirectionalIterator>& is returned. It is impossible to store things into regex_iterators. Two
end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```cpp
namespace std {
  template<class BidirectionalIterator,
           class charT = typename iterator_traits<BidirectionalIterator>::value_type,
           class traits = regex_traits<charT>>
  class regex_iterator {
    public:
    using regex_type = basic_regex<charT, traits>;
    using iterator_category = forward_iterator_tag;
    using value_type = match_results<BidirectionalIterator>;
    using difference_type = ptrdiff_t;
    using pointer = const value_type*;
    using reference = const value_type&;
    regex_iterator();
    regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                   const regex_type& re,
                   regex_constants::match_flag_type m = regex_constants::match_default);
    regex_iterator(BidirectionalIterator, BidirectionalIterator,
                   const regex_type&&,
                   regex_constants::match_flag_type = regex_constants::match_default) = delete;
    regex_iterator(const regex_iterator&);
    regex_iterator& operator=(const regex_iterator&);
    bool operator==(const regex_iterator&) const;
    const value_type& operator*() const;
    const value_type* operator->() const;
    regex_iterator& operator++();
    regex_iterator operator++(int);
  };
```

An object of type `regex_iterator` that is not an end-of-sequence iterator holds a zero-length match if `match[0].matched == true` and `match[0].first == match[0].second`.

[Note 1: For example, this can occur when the part of the regular expression that matched consists only of an assertion (such as `'^'`, `'\$'`, `'\b'`, `'\B'`). — end note]

### 30.11.1.2 Constructors

#### `regex_iterator();`

*Effects:* Constructs an end-of-sequence iterator.

#### `regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                   const regex_type& re,
                   regex_constants::match_flag_type m = regex_constants::match_default);`

*Effects:* Initializes `begin` and `end` to `a` and `b`, respectively, sets `pregex` to `addressof(re)`, sets `flags` to `m`, then calls `regex_search(begin, end, match, *pregex, flags)`. If this call returns `false` the constructor sets `*this` to the end-of-sequence iterator.

### 30.11.1.3 Comparisons

#### `bool operator==(const regex_iterator& right) const;`

*Returns:* `true` if `*this` and `right` are both end-of-sequence iterators or if the following conditions all hold:

(1.1)  

— `begin == right.begin`,

§ 30.11.3
— end == right.end,
— pregex == right.pregex,
— flags == right.flags, and
— match[0] == right.match[0];
otherwise false.

30.11.1.4 Indirection

const value_type& operator*() const;

Returns: match.

const value_type* operator->() const;

Returns: addressof(match).

30.11.1.5 Increment

regex_iterator& operator++();

Effects: Constructs a local variable start of type BidirectionalIterator and initializes it with the value of match[0].second.

If the iterator holds a zero-length match and start == end the operator sets *this to the end-of-sequence iterator and returns *this.

Otherwise, if the iterator holds a zero-length match, the operator calls:

regex_search(start, end, match, *pregex,
flags | regex_constants::match_not_null | regex_constants::match_continuous)
If the call returns true the operator returns *this. Otherwise the operator increments start and continues as if the most recent match was not a zero-length match.

If the most recent match was not a zero-length match, the operator sets flags to flags | regex_constants::match_prev_avail and calls regex_search(start, end, match, *pregex, flags). If the call returns false the iterator sets *this to the end-of-sequence iterator. The iterator then returns *this.

In all cases in which the call to regex_search returns true, match.prefix().first shall be equal to the previous value of match[0].second, and for each index i in the half-open range [0, match.size()) for which match[i].matched is true, match.position(i) shall return distance(begin, match[i].first).

[Note 1: This means that match.position(i) gives the offset from the beginning of the target sequence, which is often not the same as the offset from the sequence passed in the call to regex_search. — end note]

It is unspecified how the implementation makes these adjustments.

[Note 2: This means that a compiler can call an implementation-specific search function, in which case a program-defined specialization of regex_search will not be called. — end note]

regex_iterator operator++(int);

Effects: As if by:
regex_iterator tmp = *this;
++(*this);
return tmp;

30.11.2 Class template regex_token_iterator

30.11.2.1 General

The class template regex_token_iterator is an iterator adaptor; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more sub-expressions for each match found. Each position enumerated by the iterator is a sub_match class template instance that represents what matched a particular sub-expression within the regular expression.

§ 30.11.2.1
2 When class `regex_token_iterator` is used to enumerate a single sub-expression with index -1 the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

3 After it is constructed, the iterator finds and stores a value `regex_iterator<BidirectionalIterator>` position and sets the internal count N to zero. It also maintains a sequence subs which contains a list of the sub-expressions which will be enumerated. Every time operator++ is used the count N is incremented; if N exceeds or equals subs.size(), then the iterator increments member position and sets count N to zero.

4 If the end of sequence is reached (position is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index -1, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.

5 The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>&` is returned. The result of operator-> on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>`* is returned.

6 It is impossible to store things into `regex_token_iterator`s. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```cpp
namespace std {
    template<class BidirectionalIterator,
             class charT = typename iterator_traits<BidirectionalIterator>::value_type,
             class traits = regex_traits<charT>>
    class regex_token_iterator {
        public:
            using regex_type = basic_regex<charT, traits>;
            using iterator_category = forward_iterator_tag;
            using value_type = sub_match<BidirectionalIterator>;
            using difference_type = ptrdiff_t;
            using pointer = const value_type*;
            using reference = const value_type&;

            regex_token_iterator();
            regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                                const regex_type& re, int submatch = 0,
                                regex_constants::match_flag_type m =
                                regex_constants::match_default);
            regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                                const regex_type& re,
                                const vector<int>& submatches,
                                regex_constants::match_flag_type m =
                                regex_constants::match_default);
            regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                                const regex_type& re,
                                initializer_list<int> submatches,
                                regex_constants::match_flag_type m =
                                regex_constants::match_default);

            regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                                const regex_type&& re,
                                int submatch = 0,
                                regex_constants::match_flag_type m =
                                regex_constants::match_default) = delete;
```
A suffix iterator is a regex_token_iterator object that points to a final sequence of characters at the end of the target sequence. In a suffix iterator the member result holds a pointer to the data member suffix, the value of the member suffix.match is true, suffix.first points to the beginning of the final sequence, and suffix.second points to the end of the final sequence.

[Note 1: For a suffix iterator, data member suffix.first is the same as the end of the last match found, and suffix.second is the same as the end of the target sequence. —end note]

The current match is (*position).prefix() if subs[N] == -1, or (*position)[subs[N]] for any other value of subs[N].

### 30.11.2.2 Constructors

**Effects:** Constructs the end-of-sequence iterator.
template<size_t N>
    regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
        const regex_type& re,
        const int (&submatches)[N],
        regex_constants::match_flag_type m = regex_constants::match_default);

Preconditions: Each of the initialization values of submatches is >= -1.

Effects: The first constructor initializes the member subs to hold the single value submatch. The second, third, and fourth constructors initialize the member subs to hold a copy of the sequence of integer values pointed to by the iterator range (begin(submatches), end(submatches)).

Each constructor then sets N to 0, and position to position_iterator(a, b, re, m). If position is not an end-of-sequence iterator the constructor sets result to the address of the current match. Otherwise if any of the values stored in subs is equal to -1 the constructor sets *this to a suffix iterator that points to the range [a, b), otherwise the constructor sets *this to an end-of-sequence iterator.

### 30.11.2.3 Comparisons

**bool operator==(const regex_token_iterator& right) const;**

Returns: true if *this and right are both end-of-sequence iterators, or if *this and right are both suffix iterators and suffix == right.suffix; otherwise returns false if *this or right is an end-of-sequence iterator or a suffix iterator. Otherwise returns true if position == right.position, N == right.N, and subs == right.subs. Otherwise returns false.

### 30.11.2.4 Indirection

**const value_type& operator*() const;**

Returns: *result.

**const value_type* operator->() const;**

Returns: result.

### 30.11.2.5 Increment

**regex_token_iterator& operator++();**

Effects: Constructs a local variable prev of type position_iterator, initialized with the value of position.

If *this is a suffix iterator, sets *this to an end-of-sequence iterator.

Otherwise, if N + 1 < subs.size(), increments N and sets result to the address of the current match.

Otherwise, sets N to 0 and increments position. If position is not an end-of-sequence iterator the operator sets result to the address of the current match.

Otherwise, if any of the values stored in subs is equal to -1 and prev->suffix().length() is not 0 the operator sets *this to a suffix iterator that points to the range [prev->suffix().first, prev->suffix().second).

Otherwise, sets *this to an end-of-sequence iterator.

Returns: *this

**regex_token_iterator& operator++(int);**

Effects: Constructs a copy tmp of *this, then calls ++(*this).

Returns: tmp.
locale dependent C or C++ API, including the formatted string input functions. Instead they shall call the appropriate traits member function to achieve the required effect.

3 The following productions within the ECMAScript grammar are modified as follows:

```
ClassAtom ::
    -
    ClassAtomNoDash
    ClassAtomExClass
    ClassAtomCollatingElement
    ClassAtomEquivalence

IdentityEscape ::
    SourceCharacter but not c
```

4 The following new productions are then added:

```
ClassAtomExClass ::
    [: ClassName :]
ClassAtomCollatingElement ::
    [. ClassName .]
ClassAtomEquivalence ::
    [= ClassName =]

ClassName ::
    ClassNameCharacter
    ClassNameCharacter ClassName
    ClassNameCharacter ::
    SourceCharacter but not one of . or = or :
```

5 The productions ClassAtomExClass, ClassAtomCollatingElement and ClassAtomEquivalence provide functionality equivalent to that of the same features in regular expressions in POSIX.

6 The regular expression grammar may be modified by any regex_constants::syntax_option_type flags specified when constructing an object of type specialization of basic_regex according to the rules in Table 136.

7 A ClassName production, when used in ClassAtomExClass, is not valid if traits_inst.lookup_classname returns zero for that name. The names recognized as valid ClassNames are determined by the type of the traits class, but at least the following names shall be recognized: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit, d, s, w. In addition the following expressions shall be equivalent:

```
\d and [[:digit:]]
\D and [^[[:digit:]]]
\s and [[:space:]]
\S and [^[[:space:]]]
\w and [_[[:alnum:]]]
\W and [^[[:alnum:]]]
```

8 A ClassName production when used in a ClassAtomCollatingElement production is not valid if the value returned by traits_inst.lookup_collatename for that name is an empty string.

9 The results from multiple calls to traits_inst.lookup_classname can be bitwise OR’ed together and subsequently passed to traits_inst.isctype.

10 A ClassName production when used in a ClassAtomEquivalence production is not valid if the value returned by traits_inst.lookup_collatename for that name is an empty string or if the value returned by traits_inst.transform_primary for the result of the call to traits_inst.lookup_collatename is an empty string.

11 When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type regex_error.
If the CV of a UnicodeEscapeSequence is greater than the largest value that can be held in an object of type charT the translator shall throw an exception object of type regex_error.

[Note 1: This means that values of the form “uxxxx” that do not fit in a character are invalid. — end note]

Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling traits_inst.value.

The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMA-262. The behavior is modified according to any match_flag_type flags (30.4.3) specified when using the regular expression object in one of the regular expression algorithms (30.10). The behavior is also localized by interaction with the traits class template parameter as follows:

(14.1) — During matching of a regular expression finite state machine against a sequence of characters, two characters c and d are compared using the following rules:

(14.1.1) — if (flags() & regex_constants::icase) the two characters are equal if traits_inst.translate_nocase(c) == traits_inst.translate_nocase(d);

(14.1.2) — otherwise, if flags() & regex_constants::collate the two characters are equal if traits_inst.translate(c) == traits_inst.translate(d);

(14.1.3) — otherwise, the two characters are equal if c == d.

(14.2) — During matching of a regular expression finite state machine against a sequence of characters, comparison of a collating element range c1-c2 against a character c is conducted as follows: if flags() & regex_constants::collate is false then the character c is matched if c1 <= c && c <= c2, otherwise c is matched in accordance with the following algorithm:

```cpp
string_type str1 = string_type(1, flags() & icase ? traits_inst.translate_nocase(c1) : traits_inst.translate(c1));
string_type str2 = string_type(1, flags() & icase ? traits_inst.translate_nocase(c2) : traits_inst.translate(c2));
string_type str = string_type(1, flags() & icase ? traits_inst.translate_nocase(c) : traits_inst.translate(c));
return traits_inst.transform(str1.begin(), str1.end()) <= traits_inst.transform(str.begin(), str.end()) && traits_inst.transform(str.begin(), str.end()) <= traits_inst.transform(str2.begin(), str2.end());
```

(14.3) — During matching of a regular expression finite state machine against a sequence of characters, testing whether a collating element is a member of a primary equivalence class is conducted by first converting the collating element and the equivalence class to sort keys using traits::transform_primary, and then comparing the sort keys for equality.

(14.4) — During matching of a regular expression finite state machine against a sequence of characters, a character c is a member of a character class designated by an iterator range [first, last) if traits_inst.isctype(c, traits_inst.lookup_classname(first, last, flags() & icase)) is true.

See also: ECMA-262 15.10
31 Atomic operations library

31.1 General

1 This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.

2 The following subclauses describe atomics requirements and components for types and operations, as summarized in Table 143.

Table 143: Atomics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.3 Type aliases</td>
<td>&lt;atomic&gt;</td>
</tr>
<tr>
<td>31.4 Order and consistency</td>
<td></td>
</tr>
<tr>
<td>31.5 Lock-free property</td>
<td></td>
</tr>
<tr>
<td>31.6 Waiting and notifying</td>
<td></td>
</tr>
<tr>
<td>31.7 Class template atomic_ref</td>
<td></td>
</tr>
<tr>
<td>31.8 Class template atomic</td>
<td></td>
</tr>
<tr>
<td>31.9 Non-member functions</td>
<td></td>
</tr>
<tr>
<td>31.10 Flag type and operations</td>
<td></td>
</tr>
<tr>
<td>31.11 Fences</td>
<td></td>
</tr>
</tbody>
</table>

31.2 Header <atomic> synopsis

namespace std {
    // 31.4, order and consistency
    enum class memory_order : unspecified;
    template<class T>
        T kill_dependency(T y) noexcept;

    // 31.5, lock-free property
    #define ATOMIC_BOOL_LOCK_FREE unspecified
    #define ATOMIC_CHAR_LOCK_FREE unspecified
    #define ATOMIC_CHAR8_T_LOCK_FREE unspecified
    #define ATOMIC_CHAR16_T_LOCK_FREE unspecified
    #define ATOMIC_CHAR32_T_LOCK_FREE unspecified
    #define ATOMIC_WCHAR_T_LOCK_FREE unspecified
    #define ATOMIC_SHORT_LOCK_FREE unspecified
    #define ATOMIC_INT_LOCK_FREE unspecified
    #define ATOMIC_LONG_LOCK_FREE unspecified
    #define ATOMIC_LLONG_LOCK_FREE unspecified
    #define ATOMIC_POINTER_LOCK_FREE unspecified

    // 31.7, class template atomic_ref
    template<class T> struct atomic_ref;
    // 31.7.5, partial specialization for pointers
    template<class T> struct atomic_ref<T*>;

    // 31.8, class template atomic
    template<class T> struct atomic;
    // 31.8.5, partial specialization for pointers
    template<class T> struct atomic<T*>;

    // 31.9, non-member functions
    template<class T>
        bool atomic_is_lock_free(const volatile atomic<T>*) noexcept;
    template<class T>
        bool atomic_is_lock_free(const atomic<T>*) noexcept;
template<class T>
void atomic_store(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;

template<class T>
void atomic_store(atomic<T>*, typename atomic<T>::value_type) noexcept;

template<class T>
void atomic_store_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
void atomic_store_explicit(atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
T atomic_load(const volatile atomic<T>*) noexcept;

template<class T>
T atomic_load(const atomic<T>*) noexcept;

template<class T>
T atomic_load_explicit(const volatile atomic<T>*, memory_order) noexcept;

template<class T>
T atomic_load_explicit(const atomic<T>*, memory_order) noexcept;

template<class T>
T atomic_exchange(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_exchange(atomic<T>*, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_exchange_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
T atomic_exchange_explicit(atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
bool atomic_compare_exchange_weak(volatile atomic<T>*,
type_name atomic<T>::value_type*,
type_name atomic<T>::value_type) noexcept;

template<class T>
bool atomic_compare_exchange_weak(atomic<T>*,
type_name atomic<T>::value_type*,
type_name atomic<T>::value_type) noexcept;

template<class T>
bool atomic_compare_exchange_strong(volatile atomic<T>*,
type_name atomic<T>::value_type*,
type_name atomic<T>::value_type) noexcept;

template<class T>
bool atomic_compare_exchange_strong(atomic<T>*,
type_name atomic<T>::value_type*,
type_name atomic<T>::value_type) noexcept;

template<class T>
bool atomic_compare_exchange_weak_explicit(volatile atomic<T>*,
type_name atomic<T>::value_type*,
type_name atomic<T>::value_type,
memory_order, memory_order) noexcept;

template<class T>
bool atomic_compare_exchange_weak_explicit(atomic<T>*,
type_name atomic<T>::value_type*,
type_name atomic<T>::value_type,
memory_order, memory_order) noexcept;

template<class T>
bool atomic_compare_exchange_strong_explicit(volatile atomic<T>*,
type_name atomic<T>::value_type*,
type_name atomic<T>::value_type,
memory_order, memory_order) noexcept;

template<class T>
bool atomic_compare_exchange_strong_explicit(atomic<T>*,
type_name atomic<T>::value_type*,
type_name atomic<T>::value_type,
memory_order, memory_order) noexcept;
template<class T>
T atomic_fetch_add(volatile atomic<T>*, typename atomic<T>::difference_type) noexcept;

template<class T>
T atomic_fetch_add(atomic<T>*, typename atomic<T>::difference_type) noexcept;

template<class T>
T atomic_fetch_add_explicit(volatile atomic<T>*, typename atomic<T>::difference_type, memory_order) noexcept;

template<class T>
T atomic_fetch_add_explicit(atomic<T>*, typename atomic<T>::difference_type, memory_order) noexcept;

template<class T>
T atomic_fetch_sub(volatile atomic<T>*, typename atomic<T>::difference_type) noexcept;

template<class T>
T atomic_fetch_sub(atomic<T>*, typename atomic<T>::difference_type) noexcept;

template<class T>
T atomic_fetch_sub_explicit(volatile atomic<T>*, typename atomic<T>::difference_type, memory_order) noexcept;

template<class T>
T atomic_fetch_sub_explicit(atomic<T>*, typename atomic<T>::difference_type, memory_order) noexcept;

template<class T>
T atomic_fetch_and(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_fetch_and(atomic<T>*, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_fetch_and_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;

template<class T>
T atomic_fetch_and_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;

template<class T>
T atomic_fetch_xor(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_fetch_xor(atomic<T>*, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_fetch_xor_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;

template<class T>
T atomic_fetch_xor_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;

template<class T>
void atomic_wait(const volatile atomic<T>*, typename atomic<T>::value_type);

template<class T>
void atomic_wait(const atomic<T>*, typename atomic<T>::value_type);

template<class T>
void atomic_wait_explicit(const volatile atomic<T>*, typename atomic<T>::value_type, memory_order);

template<class T>
void atomic_wait_explicit(const atomic<T>*, typename atomic<T>::value_type, memory_order);

template<class T>
void atomic_notify_one(volatile atomic<T>**);
template<class T>
void atomic_notify_one(atomic<T>*);

template<class T>
void atomic_notify_all(volatile atomic<T>*);

template<class T>
void atomic_notify_all(atomic<T>*);

// 31.3, type aliases
using atomic_bool = atomic<bool>;
using atomic_char = atomic<char>;
using atomic_schar = atomic<signed char>;
using atomic UCHAR = atomic<unsigned char>;
using atomic_int = atomic<int>;
using atomic_uint = atomic<unsigned int>;
using atomic_long = atomic<long>;
using atomic_ulong = atomic<unsigned long>;
using atomic_int8_t = atomic<int8_t>;
using atomic_uint8_t = atomic<uint8_t>;
using atomic_int16_t = atomic<int16_t>;
using atomic_uint16_t = atomic<uint16_t>;
using atomic_int32_t = atomic<int32_t>;
using atomic_uint32_t = atomic<uint32_t>;
using atomic_int64_t = atomic<int64_t>;
using atomic_uint64_t = atomic<uint64_t>;
using atomic_int_least8_t = atomic<int_least8_t>;
using atomic_uint_least8_t = atomic<uint_least8_t>;
using atomic_int_least16_t = atomic<int_least16_t>;
using atomic_uint_least16_t = atomic<uint_least16_t>;
using atomic_int_least32_t = atomic<int_least32_t>;
using atomic_uint_least32_t = atomic<uint_least32_t>;
using atomic_int_least64_t = atomic<int_least64_t>;
using atomic_uint_least64_t = atomic<uint_least64_t>;
using atomic_int_fast8_t = atomic<int_fast8_t>;
using atomic_uint_fast8_t = atomic<uint_fast8_t>;
using atomic_int_fast16_t = atomic<int_fast16_t>;
using atomic_uint_fast16_t = atomic<uint_fast16_t>;
using atomic_int_fast32_t = atomic<int_fast32_t>;
using atomic_uint_fast32_t = atomic<uint_fast32_t>;
using atomic_intptr_t = atomic<intptr_t>;
using atomic_uintptr_t = atomic<uintptr_t>;
using atomic_size_t = atomic<size_t>;
using atomic_ptrdiff_t = atomic<ptrdiff_t>;
using atomic_intmax_t = atomic<intmax_t>;
using atomic_uintmax_t = atomic<uintmax_t>;
using atomic_int_least lock_free = see below;
using atomic_uint_least lock_free = see below;

// 31.10, flag type and operations
struct atomic_flag;
bool atomic_flag_test(const volatile atomic_flag*) noexcept;
bool atomic_flag_test(const atomic_flag*) noexcept;
bool atomic_flag_test_explicit(const volatile atomic_flag*, memory_order) noexcept;
bool atomic_flag_test_explicit(const atomic_flag*, memory_order) noexcept;
bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
bool atomic_flag_test_and_set(atomic_flag*) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
void atomic_flag_clear(volatile atomic_flag*) noexcept;
void atomic_flag_clear(atomic_flag*) noexcept;
void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order) noexcept;
void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;
void atomic_flag_wait(const volatile atomic_flag*, bool) noexcept;
void atomic_flag_wait(const atomic_flag*, bool) noexcept;
void atomic_flag_wait_explicit(const volatile atomic_flag*, bool, memory_order) noexcept;
void atomic_flag_wait_explicit(const atomic_flag*, bool, memory_order) noexcept;
void atomic_flag_notify_one(volatile atomic_flag*) noexcept;
void atomic_flag_notify_one(atomic_flag*) noexcept;
void atomic_flag_notify_all(volatile atomic_flag*) noexcept;
void atomic_flag_notify_all(atomic_flag*) noexcept;

// 31.11. fences
extern "C" void atomic_thread_fence(memory_order) noexcept;
extern "C" void atomic_signal_fence(memory_order) noexcept;

31.3 Type aliases [atomics.alias]

1 The type aliases atomic_intN_t, atomic_uintN_t, atomic_intptr_t, and atomic_uintptr_t are defined if and only if intN_t, uintN_t, intptr_t, and uintptr_t are defined, respectively.

2 The type aliases atomic_signed_lock_free and atomic_unsigned_lock_free name specializations of atomic whose template arguments are integral types, respectively signed and unsigned, and whose is_always_lock_free property is true.

[Note 1: These aliases are optional in freestanding implementations (16.4.2.4). — end note]

Implementations should choose for these aliases the integral specializations of atomic for which the atomic waiting and notifying operations (31.6) are most efficient.

31.4 Order and consistency [atomics.order]

namespace std {
    enum class memory_order : unspecified {
        relaxed, consume, acquire, release, acq_rel, seq_cst
    };
    inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
    inline constexpr memory_order memory_order_consume = memory_order::consume;
    inline constexpr memory_order memory_order_acquire = memory_order::acquire;
    inline constexpr memory_order memory_order_release = memory_order::release;
    inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
    inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
}

1 The enumeration memory_order specifies the detailed regular (non-atomic) memory synchronization order as defined in 6.9.2 and may provide for operation ordering. Its enumerated values and their meanings are as follows:

   — memory_order::relaxed: no operation orders memory.
   — memory_order::release, memory_order::acq_rel, and memory_order::seq_cst: a store operation performs a release operation on the affected memory location.
   — memory_order::consume: a load operation performs a consume operation on the affected memory location.
§ 31.4 1544

There is a single total order $S$ on all \texttt{memory\_order::seq\_cst} operations, including fences, that satisfies the following constraints. First, if $A$ and $B$ are \texttt{memory\_order::seq\_cst} operations and $A$ strongly happens before $B$, then $A$ precedes $B$ in $S$. Second, for every pair of atomic operations $A$ and $B$ on an object $M$, where $A$ is coherence-ordered before $B$, the following four conditions are required to be satisfied by $S$:

- $A$ is a modification, and $B$ reads the value stored by $A$, or
- $A$ precedes $B$ in the modification order of $M$, or
- $A$ and $B$ are not the same atomic read-modify-write operation, and there exists an atomic modification $X$ of $M$ such that $A$ reads the value stored by $X$ and $X$ precedes $B$ in the modification order of $M$, or
- there exists an atomic modification $X$ of $M$ such that $A$ is coherence-ordered before $X$ and $X$ is coherence-ordered before $B$.

There are implementations that use either \texttt{memory\_order::seq\_cst} or \texttt{memory\_order::acquire} for atomic operations. The specification ensures that the implementation of \texttt{memory\_order::seq\_cst} ensures sequential consistency only for a program that is free of data races and coherence-ordered before any modification of $M$ that precedes $A$ in $S$. — end note]}

Note 4: We do not require that $S$ be consistent with “happens before” (6.9.2.2). This allows more efficient implementation of \texttt{memory\_order::acquire} and \texttt{memory\_order::release} on some machine architectures. It can produce surprising results when these are mixed with \texttt{memory\_order::seq\_cst} accesses. — end note]

Note 5: \texttt{memory\_order::seq\_cst} ensures sequential consistency only for a program that is free of data races and uses exclusively \texttt{memory\_order::seq\_cst} atomic operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In many cases, \texttt{memory\_order::seq\_cst} atomic operations are reorderable with respect to other atomic operations performed by the same thread. — end note]

Implementations should ensure that no “out-of-thin-air” values are computed that circularly depend on their own computation.

Note 6: For example, with $x$ and $y$ initially zero,

```c
// Thread 1:
x.store(r1, memory_order::relaxed);
// Thread 2:
y.store(r2, memory_order::relaxed);
```

this recommendation discourages producing $r1 == r2 == 42$, since the store of 42 to $y$ is only possible if the store to $x$ stores 42, which circularly depends on the store to $y$ storing 42. Note that without this restriction, such an execution is possible. — end note]}

© ISO/IEC N4868

Implementations should ensure that no “out-of-thin-air” values are computed that circularly depend on their own computation.

Note 6: For example, with $x$ and $y$ initially zero,

```c
// Thread 1:
x.store(r1, memory_order::relaxed);
// Thread 2:
y.store(r2, memory_order::relaxed);
```

this recommendation discourages producing $r1 == r2 == 42$, since the store of 42 to $y$ is only possible if the store to $x$ stores 42, which circularly depends on the store to $y$ storing 42. Note that without this restriction, such an execution is possible. — end note]}

© ISO/IEC N4868

Implementations should ensure that no “out-of-thin-air” values are computed that circularly depend on their own computation.

Note 6: For example, with $x$ and $y$ initially zero,

```c
// Thread 1:
x.store(r1, memory_order::relaxed);
// Thread 2:
y.store(r2, memory_order::relaxed);
```

this recommendation discourages producing $r1 == r2 == 42$, since the store of 42 to $y$ is only possible if the store to $x$ stores 42, which circularly depends on the store to $y$ storing 42. Note that without this restriction, such an execution is possible. — end note]}

© ISO/IEC N4868

Implementations should ensure that no “out-of-thin-air” values are computed that circularly depend on their own computation.

Note 6: For example, with $x$ and $y$ initially zero,

```c
// Thread 1:
x.store(r1, memory_order::relaxed);
// Thread 2:
y.store(r2, memory_order::relaxed);
```

this recommendation discourages producing $r1 == r2 == 42$, since the store of 42 to $y$ is only possible if the store to $x$ stores 42, which circularly depends on the store to $y$ storing 42. Note that without this restriction, such an execution is possible. — end note]
9 [Note 7: The recommendation similarly disallows \( r1 == r2 == 42 \) in the following example, with \( x \) and \( y \) again initially zero:]

```c
// Thread 1:
int r1 = x.load(memory_order::relaxed);
if (r1 == 42) y.store(42, memory_order::relaxed);
// Thread 2:
int r2 = y.load(memory_order::relaxed);
if (r2 == 42) x.store(42, memory_order::relaxed);
@endnote]
```

Atomic read-modify-write operations shall always read the last value (in the modification order) written before the write associated with the read-modify-write operation.

Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

```c
template<class T>
T kill_dependency(T y) noexcept;
```

10 Effects: The argument does not carry a dependency to the return value (6.9.2).

Returns: \( y \).

### 31.5 Lock-free property

The \texttt{ATOMIC\_..\_LOCK\_FREE} macros indicate the lock-free property of the corresponding atomic types, with the signed and unsigned variants grouped together. The properties also apply to the corresponding (partial) specializations of the \texttt{atomic} template. A value of 0 indicates that the types are never lock-free. A value of 1 indicates that the types are sometimes lock-free. A value of 2 indicates that the types are always lock-free.

At least one signed integral specialization of the \texttt{atomic} template, along with the specialization for the corresponding unsigned type (6.8.2), is always lock-free.

[Note 1: This requirement is optional in freestanding implementations (16.4.2.4). — end note]

The function \texttt{atomic\_is\_lock\_free} (31.8.2) indicates whether the object is lock-free. In any given program execution, the result of the lock-free query shall be consistent for all pointers of the same type.

Atomic operations that are not lock-free are considered to potentially block (6.9.2.3).

Recommended practice: Operations that are lock-free should also be address-free\(^{328}\). The implementation of these operations should not depend on any per-process state.

[Note 2: This restriction enables communication by memory that is mapped into a process more than once and by memory that is shared between two processes. — end note]

### 31.6 Waiting and notifying

Atomic waiting operations and atomic notifying operations provide a mechanism to wait for the value of an atomic object to change more efficiently than can be achieved with polling. An atomic waiting operation may block until it is unblocked by an atomic notifying operation, according to each function's effects.

[Note 1: Programs are not guaranteed to observe transient atomic values, an issue known as the A-B-A problem, resulting in continued blocking if a condition is only temporarily met. — end note]

[Note 2: The following functions are atomic waiting operations:

(2.1) ```c
        — atomic\langle T\rangle::wait,
```]

\(^{328}\) That is, atomic operations on the same memory location via two different addresses will communicate atomically.
atomic_flag::wait, atomic_wait and atomic_wait_explicit, atomic_flag_wait and atomic_flag_wait_explicit, and atomic_flag::wait.

— end note

— end note

3 [Note 3: The following functions are atomic notifying operations:

atomic<T>::notify_one and atomic<T>::notify_all,

atomic_flag::notify_one and atomic_flag::notify_all,

atomic_notify_one and atomic_notify_all,

atomic_flag_notify_one and atomic_flag_notify_all,

atomic_ref<T>::notify_one and atomic_ref<T>::notify_all.

— end note

4 A call to an atomic waiting operation on an atomic object M is eligible to be unblocked by a call to an atomic notifying operation on M if there exist side effects X and Y on M such that:

— the atomic waiting operation has blocked after observing the result of X,
— X precedes Y in the modification order of M, and
— Y happens before the call to the atomic notifying operation.

31.7 Class template atomic_ref

31.7.1 General

namespace std {
    template<class T> struct atomic_ref {
        private:
            T* ptr;  // exposition only
        public:
            using value_type = T;
            static constexpr size_t required_alignment = implementation-defined;
            static constexpr bool is_always_lock_free = implementation-defined;
            bool is_lock_free() const noexcept;
            explicit atomic_ref(T&);
            atomic_ref(const atomic_ref&) noexcept;
            atomic_ref& operator=(const atomic_ref&) = delete;
            void store(T, memory_order = memory_order::seq_cst) const noexcept;
            T load(memory_order = memory_order::seq_cst) const noexcept;
            operator T() const noexcept;
            T exchange(T, memory_order = memory_order::seq_cst) const noexcept;
            bool compare_exchange_weak(T&, T,
                memory_order, memory_order) const noexcept;
            bool compare_exchange_strong(T&, T,
                memory_order, memory_order) const noexcept;
            bool compare_exchange_weak(T&, T,
                memory_order = memory_order::seq_cst) const noexcept;
            bool compare_exchange_strong(T&, T,
                memory_order = memory_order::seq_cst) const noexcept;
            void wait(T, memory_order = memory_order::seq_cst) const noexcept;
            void notify_one() const noexcept;
            void notify_all() const noexcept;
    };

};
An `atomic_ref` object applies atomic operations (31.1) to the object referenced by `*ptr` such that, for the lifetime (6.7.3) of the `atomic_ref` object, the object referenced by `*ptr` is an atomic object (6.9.2.2).

The program is ill-formed if `is_trivially_copyable_v<T>` is `false`.

The lifetime (6.7.3) of an object referenced by `*ptr` shall exceed the lifetime of all `atomic_ref`s that reference the object. While any `atomic_ref` instances exist that reference the `*ptr` object, all accesses to that object shall exclusively occur through those `atomic_ref` instances. No subobject of the object referenced by `atomic_ref` shall be concurrently referenced by any other `atomic_ref` object.

Atomic operations applied to an object through a referencing `atomic_ref` are atomic with respect to atomic operations applied through any other `atomic_ref` referencing the same object.

[Note 1: Atomic operations or the `atomic_ref` constructor could acquire a shared resource, such as a lock associated with the referenced object, to enable atomic operations to be applied to the referenced object. — end note]

### 31.7.2 Operations

```c++
static constexpr size_t required_alignment;
```

1. The alignment required for an object to be referenced by an atomic reference, which is at least `alignof(T)`.

2. [Note 1: Hardware could require an object referenced by an `atomic_ref` to have stricter alignment (6.7.6) than other objects of type `T`. Further, whether operations on an `atomic_ref` are lock-free could depend on the alignment of the referenced object. For example, lock-free operations on `std::complex<double>` could be supported only if aligned to `2*alignof(double)`. — end note]

```c++
static constexpr bool is_always_lock_free;
```

3. The static data member `is_always_lock_free` is `true` if the `atomic_ref` type’s operations are always lock-free, and `false` otherwise.

```c++
bool is_lock_free() const noexcept;
```

4. Returns: `true` if operations on all objects of the type `atomic_ref<T>` are lock-free, `false` otherwise.

```c++
atomic_ref(T& obj);
```

5. Preconditions: The referenced object is aligned to `required_alignment`.

6. Postconditions: `*this` references `obj`.


```c++
atomic_ref(const atomic_ref& ref) noexcept;
```

8. Postconditions: `*this` references the object referenced by `ref`.

```c++
void store(T desired, memory_order order = memory_order::seq_cst) const noexcept;
```

9. Preconditions: The `order` argument is neither `memory_order::consume`, `memory_order::acquire`, nor `memory_order::acq_rel`.

10. Effects: Atomically replaces the value referenced by `*ptr` with the value of `desired`. Memory is affected according to the value of `order`.

```c++
T operator=(T desired) const noexcept;
```

11. Effects: Equivalent to:

    ```c++
    store(desired);
    return desired;
    ```

```c++
T load(memory_order order = memory_order::seq_cst) const noexcept;
```

12. Preconditions: The `order` argument is neither `memory_order::release` nor `memory_order::acq_rel`.

13. Effects: Memory is affected according to the value of `order`.

14. Returns: Atomically returns the value referenced by `*ptr`.

```c++
operator T() const noexcept;
```

15. Effects: Equivalent to: `return load();`

§ 31.7.2 1547
T exchange(T desired, memory_order order = memory_order::seq_cst) const noexcept;

Effects: Atomically replaces the value referenced by *ptr with desired. Memory is affected according to the value of order. This operation is an atomic read-modify-write operation (6.9.2).

Returns: Atomically returns the value referenced by *ptr immediately before the effects.

bool compare_exchange_weak(T& expected, T desired, memory_order success, memory_order failure) const noexcept;

bool compare_exchange_strong(T& expected, T desired, memory_order success, memory_order failure) const noexcept;

bool compare_exchange_weak(T& expected, T desired, memory_order order = memory_order::seq_cst) const noexcept;

bool compare_exchange_strong(T& expected, T desired, memory_order order = memory_order::seq_cst) const noexcept;

Preconditions: The failure argument is neither memory_order::release nor memory_order::acq_rel.

Effects: Retrieves the value in expected. It then atomically compares the value representation of the value referenced by *ptr for equality with that previously retrieved from expected, and if true, replaces the value referenced by *ptr with that in desired. If and only if the comparison is true, memory is affected according to the value of success, and if the comparison is false, memory is affected according to the value of failure. When only one memory_order argument is supplied, the value of success is order, and the value of failure is order except that a value of memory_order::acq_rel shall be replaced by the value memory_order::acquire and a value of memory_order::release shall be replaced by the value memory_order::relaxed. If and only if the comparison is false then, after the atomic operation, the value in expected is replaced by the value read from the value referenced by *ptr during the atomic comparison. If the operation returns true, these operations are atomic read-modify-write operations (6.9.2.2) on the value referenced by *ptr. Otherwise, these operations are atomic load operations on that memory.

Returns: The result of the comparison.

Remarks: A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by expected and ptr are equal, it may return false and store back to expected the same memory contents that were originally there.

[Note 2: This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines. A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop. When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable. — end note]

void wait(T old, memory_order order = memory_order::seq_cst) const noexcept;

Preconditions: order is neither memory_order::release nor memory_order::acq_rel.

Effects: Repeatedly performs the following steps, in order:

(23.1) Evaluates load(order) and compares its value representation for equality against that of old.

(23.2) If they compare unequal, returns.

(23.3) Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.

Remarks: This function is an atomic waiting operation (31.6) on atomic object *ptr.

void notify_one() const noexcept;

Effects: Unblocks the execution of at least one atomic waiting operation on *ptr that is eligible to be unblocked (31.6) by this call, if any such atomic waiting operations exist.

Remarks: This function is an atomic notifying operation (31.6) on atomic object *ptr.
void notify_all() const noexcept;

Effects: Unblocks the execution of all atomic waiting operations on *ptr that are eligible to be unblocked (31.6) by this call.

Remarks: This function is an atomic notifying operation (31.6) on atomic object *ptr.

31.7.3 Specializations for integral types [atomics.ref.int]

There are specializations of the atomic_ref class template for the integral types char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, char8_t, char16_t, char32_t, wchar_t, and any other types needed by the typedefs in the header <cstdint> (17.4.2). For each such type integral, the specialization atomic_ref<integral> provides additional atomic operations appropriate to integral types.

[Note 1: The specialization atomic_ref<bool> uses the primary template (31.7). — end note]

namespace std {
    template<> struct atomic_ref<integral> {
        private:
            integral* ptr; // exposition only
        public:
            using value_type = integral;
            using difference_type = value_type;
            static constexpr size_t required_alignment = implementation-defined;
            static constexpr bool is_always_lock_free = implementation-defined;
            bool is_lock_free() const noexcept;
            explicit atomic_ref(integral&);
            atomic_ref(const atomic_ref&) noexcept;
            atomic_ref& operator=(const atomic_ref&) = delete;
            void store(integral, memory_order = memory_order::seq_cst) const noexcept;
            integral operator=(const atomic_ref&) = delete;
            integral load(memory_order = memory_order::seq_cst) const noexcept;
            operator integral() const noexcept;
            integral exchange(integral,
                memory_order = memory_order::seq_cst) const noexcept;
            bool compare_exchange_weak(integral&, integral,
                memory_order, memory_order) const noexcept;
            bool compare_exchange_strong(integral&, integral,
                memory_order, memory_order) const noexcept;
            bool compare_exchange_weak(integral&, integral,
                memory_order = memory_order::seq_cst) const noexcept;
            bool compare_exchange_strong(integral&, integral,
                memory_order = memory_order::seq_cst) const noexcept;
            integral fetch_add(integral,
                memory_order = memory_order::seq_cst) const noexcept;
            integral fetch_sub(integral,
                memory_order = memory_order::seq_cst) const noexcept;
            integral fetch_and(integral,
                memory_order = memory_order::seq_cst) const noexcept;
            integral fetch_or(integral,
                memory_order = memory_order::seq_cst) const noexcept;
            integral fetch_xor(integral,
                memory_order = memory_order::seq_cst) const noexcept;
            integral operator++(int) const noexcept;
            integral operator--(int) const noexcept;
            integral operator++() const noexcept;
            integral operator--() const noexcept;
            integral operator+=(integral) const noexcept;
            integral operator-=(integral) const noexcept;
    }
};
integral operator&=(integral) const noexcept;
integral operator|=(integral) const noexcept;
integral operator^=(integral) const noexcept;

void wait(integral, memory_order = memory_order::seq_cst) const noexcept;
void notify_one() const noexcept;
void notify_all() const noexcept;
};

Descriptions are provided below only for members that differ from the primary template.

The following operations perform arithmetic computations. The key, operator, and computation correspondence is identified in Table 144.

integral fetch_key(integral operand, memory_order order = memory_order::seq_cst) const noexcept;

Effects: Atomically replaces the value referenced by *ptr with the result of the computation applied to the value referenced by *ptr and the given operand. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (6.9.2.2).

Returns: Atomically, the value referenced by *ptr immediately before the effects.

Remarks: For signed integer types, the result is as if the object value and parameters were converted to their corresponding unsigned types, the computation performed on those types, and the result converted back to the signed type.

[Note 2: There are no undefined results arising from the computation. —end note]

integral operator op=(integral operand) const noexcept;

Effects: Equivalent to: return fetch_key(operand) op operand;

§ 31.7.4 Specializations for floating-point types

There are specializations of the atomic_ref class template for the floating-point types float, double, and long double. For each such type floating-point, the specialization atomic_ref<floating-point> provides additional atomic operations appropriate to floating-point types.

namespace std {
    template<> struct atomic_ref<
        template<> struct atomic_ref<
        
        using value_type = floating-point;
        using difference_type = value_type;
        static constexpr size_t required_alignment = implementation-defined;
        static constexpr bool is_always_lock_free = implementation-defined;
        bool is_lock_free() const noexcept;
        explicit atomic_ref(floating-point);
        atomic_ref(const atomic_ref&) noexcept;
        atomic_ref& operator=(const atomic_ref&) = delete;
        void store(floating-point, memory_order = memory_order::seq_cst) const noexcept;
        floating-point operator=(floating-point) const noexcept;
        floating-point load(memory_order = memory_order::seq_cst) const noexcept;
        operator floating-point() const noexcept;
        floating-point exchange(floating-point,
            memory_order = memory_order::seq_cst) const noexcept;
        bool compare_exchange_weak(floating-point, floating-point,
            memory_order, memory_order) const noexcept;
        bool compare_exchange_strong(floating-point, floating-point,
            memory_order, memory_order) const noexcept;
        bool compare_exchange_weak(floating-point, floating-point,
            memory_order = memory_order::seq_cst) const noexcept;

§ 31.7.4
bool compare_exchange_strong(floating-point&, floating-point,
    memory_order = memory_order::seq_cst) const noexcept;

floating-point fetch_add(floating-point,
    memory_order = memory_order::seq_cst) const noexcept;

floating-point fetch_sub(floating-point,
    memory_order = memory_order::seq_cst) const noexcept;

floating-point operator+=(floating-point) const noexcept;
floating-point operator-=(floating-point) const noexcept;

void wait(floating-point, memory_order = memory_order::seq_cst) const noexcept;
void notify_one() const noexcept;
void notify_all() const noexcept;
}

2 Descriptions are provided below only for members that differ from the primary template.

3 The following operations perform arithmetic computations. The key, operator, and computation correspondence are identified in Table 144.

4 Effects: Atomically replaces the value referenced by *ptr with the result of the computation applied to the value referenced by *ptr and the given operand. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (6.9.2.2).

5 Returns: Atomically, the value referenced by *ptr immediately before the effects.

6 Remarks: If the result is not a representable value for its type (7.1), the result is unspecified, but the operations otherwise have no undefined behavior. Atomic arithmetic operations on floating-point should conform to the std::numeric_limits<floating-point> traits associated with the floating-point type (17.3.3). The floating-point environment (26.3) for atomic arithmetic operations on floating-point may be different than the calling thread’s floating-point environment.

7 Effects: Equivalent to: return fetch_key(operand) op operand;

### 31.7.5 Partial specialization for pointers

```cpp
namespace std {
    template<class T> struct atomic_ref<T*> {
        private:
            T** ptr; // exposition only
        public:
            using value_type = T*;
            using difference_type = ptrdiff_t;
            static constexpr size_t required_alignment = implementation-defined;
            static constexpr bool is_always_lock_free = implementation-defined;
            static constexpr bool is_lock_free() const noexcept;
            explicit atomic_ref(T*);
            atomic_ref(const atomic_ref&) noexcept;
            atomic_ref& operator=(const atomic_ref&) = delete;
            void store(T*, memory_order = memory_order::seq_cst) const noexcept;
            T* operator=(T*) const noexcept;
            T* load(memory_order = memory_order::seq_cst) const noexcept;
            operator T*() const noexcept;
            T* exchange(T*, memory_order = memory_order::seq_cst) const noexcept;
            bool compare_exchange_weak(T*, T*,
                                           memory_order, memory_order) const noexcept;
    };
```
bool compare_exchange_strong(T*&, T*, memory_order, memory_order) const noexcept;
bool compare_exchange_weak(T*&, T*, memory_order = memory_order::seq_cst) const noexcept;
bool compare_exchange_strong(T*&, T*, memory_order = memory_order::seq_cst) const noexcept;

T* fetch_add(difference_type, memory_order = memory_order::seq_cst) const noexcept;
T* fetch_sub(difference_type, memory_order = memory_order::seq_cst) const noexcept;

T* operator++(int) const noexcept;
T* operator--(int) const noexcept;
T* operator++() const noexcept;
T* operator--() const noexcept;
T* operator+=(difference_type) const noexcept;
T* operator-=(difference_type) const noexcept;

void wait(T*, memory_order = memory_order::seq_cst) const noexcept;
void notify_one() const noexcept;
void notify_all() const noexcept;

}
static constexpr bool is_always_lock_free = implementation-defined;
bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;

// 31.8.2, operations on atomic types
constexpr atomic() noexcept(is_nothrow_default_constructible_v<T>);
constexpr atomic(T) noexcept;
atomic(const atomic&) = delete;
atomic& operator=(const atomic&) = delete;
atomic& operator=(const atomic&) volatile = delete;
T load(memory_order = memory_order::seq_cst) const volatile noexcept;
T load(memory_order = memory_order::seq_cst) const noexcept;
operator T() const volatile noexcept;
operator T() const noexcept;
void store(T, memory_order = memory_order::seq_cst) volatile noexcept;
void store(T, memory_order = memory_order::seq_cst) noexcept;
T operator=(T) volatile noexcept;
T operator=(T) noexcept;
T exchange(T, memory_order = memory_order::seq_cst) volatile noexcept;
T exchange(T, memory_order = memory_order::seq_cst) noexcept;
bool compare_exchange_weak(T&, T, memory_order, memory_order) volatile noexcept;
bool compare_exchange_weak(T&, T, memory_order, memory_order) noexcept;
bool compare_exchange_strong(T&, T, memory_order, memory_order) volatile noexcept;
bool compare_exchange_strong(T&, T, memory_order, memory_order) noexcept;
bool compare_exchange_weak(T&, T, memory_order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_weak(T&, T, memory_order = memory_order::seq_cst) noexcept;
bool compare_exchange_strong(T&, T, memory_order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_strong(T&, T, memory_order = memory_order::seq_cst) noexcept;
void wait(T, memory_order = memory_order::seq_cst) const volatile noexcept;
void wait(T, memory_order = memory_order::seq_cst) const noexcept;
void notify_one() volatile noexcept;
void notify_one() noexcept;
void notify_all() volatile noexcept;
void notify_all() noexcept;
};

1 The template argument for T shall meet the Cpp17CopyConstructible and Cpp17CopyAssignable requirements. The program is ill-formed if any of
(1.1) — is_trivially_copyable_v<T>,
(1.2) — is_copy_constructible_v<T>,
(1.3) — is_move_constructible_v<T>,
(1.4) — is_copyAssignable_v<T>, or
(1.5) — is_move_assignable_v<T> is false.
[Note 1: Type arguments that are not also statically initializable might be difficult to use. — end note]
2 The specialization atomic<bool> is a standard-layout struct.
3 [Note 2: The representation of an atomic specialization need not have the same size and alignment requirement as its corresponding argument type. — end note]

31.8.2 Operations on atomic types
[atomics.types.operations]
constexpr atomic() noexcept(is_nothrow_default_constructible_v<T>);

1 Mandates: is_default_constructible_v<T> is true.
2 Effects: Initializes the atomic object with the value of T(). Initialization is not an atomic operation (6.9.2).
constexpr atomic(T desired) noexcept;

Effects: Initializes the object with the value desired. Initialization is not an atomic operation (6.9.2).

[Note 1: It is possible to have an access to an atomic object A race with its construction, for example by communicating the address of the just-constructed object A to another thread via memory_order::relaxed operations on a suitable atomic pointer variable, and then immediately accessing A in the receiving thread. This results in undefined behavior. — end note]

static constexpr bool is_always_lock_free = implementation-defined;

The static data member is_always_lock_free is true if the atomic type’s operations are always lock-free, and false otherwise.

[Note 2: The value of is_always_lock_free is consistent with the value of the corresponding ATOMIC_..._LOCK_FREE macro, if defined. — end note]

bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;

Returns: true if the object’s operations are lock-free, false otherwise.

[Note 3: The return value of the is_lock_free member function is consistent with the value of is_always_lock_free for the same type. — end note]

void store(T desired, memory_order order = memory_order::seq_cst) volatile noexcept;
void store(T desired, memory_order order = memory_order::seq_cst) noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Preconditions: The order argument is neither memory_order::consume, memory_order::acquire, nor memory_order::acq_rel.

Effects: Atomically replaces the value pointed to by this with the value of desired. Memory is affected according to the value of order.

T operator=(T desired) volatile noexcept;
T operator=(T desired) noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Effects: Equivalent to store(desired).

Returns: desired.

T load(memory_order order = memory_order::seq_cst) const volatile noexcept;
T load(memory_order order = memory_order::seq_cst) const noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Preconditions: The order argument is neither memory_order::release nor memory_order::acq_rel.

Effects: Memory is affected according to the value of order.

Returns: Atomically returns the value pointed to by this.

operator T() const volatile noexcept;
operator T() const noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Effects: Equivalent to: return load();

T exchange(T desired, memory_order order = memory_order::seq_cst) volatile noexcept;
T exchange(T desired, memory_order order = memory_order::seq_cst) noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Effects: Atomically replaces the value pointed to by this with desired. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (6.9.2).

Returns: Atomically returns the value pointed to by this immediately before the effects.

bool compare_exchange_weak(T& expected, T desired,
memory_order success, memory_order failure) volatile noexcept;
bool compare_exchange_weak(T& expected, T desired,
    memory_order success, memory_order failure) noexcept;
bool compare_exchange_strong(T& expected, T desired,
    memory_order success, memory_order failure) volatile noexcept;
bool compare_exchange_strong(T& expected, T desired,
    memory_order success, memory_order failure) noexcept;
bool compare_exchange_weak(T& expected, T desired,
    memory_order order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_weak(T& expected, T desired,
    memory_order order = memory_order::seq_cst) noexcept;
bool compare_exchange_strong(T& expected, T desired,
    memory_order order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_strong(T& expected, T desired,
    memory_order order = memory_order::seq_cst) noexcept;

21 Constraints: For the volatile overload of this function, is_always_lock_free is true.
22 Preconditions: The failure argument is neither memory_order::release nor memory_order::acq_rel.
23 Effects: Retrieves the value in expected. It then atomically compares the value representation of
the value pointed to by this for equality with that previously retrieved from expected, and if true,
replaces the value pointed to by this with that in desired. If and only if the comparison is true,
memory is affected according to the value of success, and if the comparison is false, memory is affected
according to the value of failure. When only one memory_order argument is supplied, the value of
success is order, and the value of failure is order except that a value of memory_order::acq_rel
shall be replaced by the value memory_order::acquire and a value of memory_order::release shall
be replaced by the value memory_order::relaxed. If and only if the comparison is false then, after
the atomic operation, the value in expected is replaced by the value pointed to by this during the
atomic comparison. If the operation returns true, these operations are atomic read-modify-write
operations (6.9.2) on the memory pointed to by this. Otherwise, these operations are atomic load
operations on that memory.
24 Returns: The result of the comparison.
25 [Note 4: For example, the effect of compare_exchange_strong on objects without padding bits (6.8) is

    if (memcmp(this, &expected, sizeof(*this)) == 0)
        memcpy(this, &desired, sizeof(*this));
    else
        memcpy(&expected, this, sizeof(*this));
    —end note]
26 [Example 1: The expected use of the compare-and-exchange operations is as follows. The compare-and-exchange
operations will update expected when another iteration of the loop is needed.

    expected = current.load();
    do {
        desired = function(expected);
    } while (!current.compare_exchange_weak(expected, desired));
    —end example]
27 [Example 2: Because the expected value is updated only on failure, code releasing the memory containing
the expected value on success will work. For example, list head insertion will act atomically and would not
introduce a data race in the following code:

    do {
        p->next = head;            // make new list node point to the current head
    } while (!head.compare_exchange_weak(p->next, p)); // try to insert
    —end example]
28 Implementations should ensure that weak compare-and-exchange operations do not consistently return
false unless either the atomic object has value different from expected or there are concurrent
modifications to the atomic object.
29 Remarks: A weak compare-and-exchange operation may fail spuriously. That is, even when the contents
of memory referred to by expected and this are equal, it may return false and store back to expected
the same memory contents that were originally there.
[Note 5: This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines. A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop. When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable. — end note]

[Note 6: Under cases where the memcpy and memcmp semantics of the compare-and-exchange operations apply, the outcome might be failed comparisons for values that compare equal with operator== if the value representation has trap bits or alternate representations of the same value. Notably, on implementations conforming to ISO/IEC/IEEE 60559, floating-point ~0.0 and +0.0 will not compare equal with memcmp but will compare equal with operator==, and NaNs with the same payload will compare equal with memcmp but will not compare equal with operator==. — end note]

[Note 7: Because compare-and-exchange acts on an object’s value representation, padding bits that never participate in the object’s value representation are ignored. As a consequence, the following code is guaranteed to avoid spurious failure:

```c
struct padded {
    char clank = 0x42;
    // Padding here.
    unsigned biff = 0xCODEFEEF;
};
atomic<padded> pad = {};

bool zap() {
    padded expected, desired{0, 0};
    return pad.compare_exchange_strong(expected, desired);
}
— end note]

[Note 8: For a union with bits that participate in the value representation of some members but not others, compare-and-exchange might always fail. This is because such padding bits have an indeterminate value when they do not participate in the value representation of the active member. As a consequence, the following code is not guaranteed to ever succeed:

```c
union pony {
    double celestia = 0.;
    short luna;       // padded
};
atomic<pony> princesses = {};

bool party(pony desired) {
    pony expected;
    return princesses.compare_exchange_strong(expected, desired);
}
— end note]

void wait(T old, memory_order order = memory_order::seq_cst) const volatile noexcept;
void wait(T old, memory_order order = memory_order::seq_cst) const noexcept;

Preconditions: order is neither memory_order::release nor memory_order::acq_rel.

Effects: Repeatedly performs the following steps, in order:

(30.1) — Evaluates load(order) and compares its value representation for equality against that of old.

(30.2) — If they compare unequal, returns.

(30.3) — Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.

Remarks: This function is an atomic waiting operation (31.6).

void notify_one() volatile noexcept;
void notify_one() noexcept;

Effects: Unblocks the execution of at least one atomic waiting operation that is eligible to be unblocked (31.6) by this call, if any such atomic waiting operations exist.

Remarks: This function is an atomic notifying operation (31.6).
void notify_all() volatile noexcept;
void notify_all() noexcept;

Effects: Unblocks the execution of all atomic waiting operations that are eligible to be unblocked (31.6) by this call.

Remarks: This function is an atomic notifying operation (31.6).

31.8.3 Specializations for integers

There are specializations of the atomic class template for the integral types char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, char8_t, char16_t, char32_t, wchar_t, and any other types needed by the typedefs in the header <cstdint> (17.4.2). For each such type integral, the specialization atomic<integral> provides additional atomic operations appropriate to integral types.

[Note 1: The specialization atomic<bool> uses the primary template (31.8). — end note]

namespace std {
    template<> struct atomic<integral> {
        using value_type = integral;
        using difference_type = value_type;

        static constexpr bool is_always_lock_free = implementation-defined;
        bool is_lock_free() const volatile noexcept;
        bool is_lock_free() const noexcept;

        atomic() noexcept;
        atomic(integral) noexcept;
        void store(integral, memory_order = memory_order::seq_cst) volatile noexcept;
        void store(integral, memory_order = memory_order::seq_cst) noexcept;
        integral operator=(integral) volatile noexcept;
        integral operator=(integral) noexcept;
        integral load(memory_order = memory_order::seq_cst) const volatile noexcept;
        integral load(memory_order = memory_order::seq_cst) const noexcept;
        operator integral() const volatile noexcept;
        operator integral() const noexcept;
        integral exchange(integral, memory_order = memory_order::seq_cst) volatile noexcept;
        integral exchange(integral, memory_order = memory_order::seq_cst) noexcept;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order, memory_order) volatile noexcept;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order, memory_order) noexcept;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order, memory_order) volatile noexcept;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order, memory_order) noexcept;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order = memory_order::seq_cst) volatile noexcept;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order = memory_order::seq_cst) noexcept;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order = memory_order::seq_cst) volatile noexcept;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order = memory_order::seq_cst) noexcept;
        integral fetch_add(integral, memory_order = memory_order::seq_cst) volatile noexcept;
        integral fetch_add(integral, memory_order = memory_order::seq_cst) noexcept;
        integral fetch_sub(integral, memory_order = memory_order::seq_cst) volatile noexcept;
        integral fetch_sub(integral, memory_order = memory_order::seq_cst) noexcept;
        integral fetch_and(integral, memory_order = memory_order::seq_cst) volatile noexcept;
        integral fetch_and(integral, memory_order = memory_order::seq_cst) noexcept;
    }
}
integral fetch_and(integral, memory_order = memory_order::seq_cst) noexcept;
integral fetch_or(integral, memory_order = memory_order::seq_cst) volatile noexcept;
integral fetch_or(integral, memory_order = memory_order::seq_cst) noexcept;
integral fetch_xor(integral, memory_order = memory_order::seq_cst) volatile noexcept;
integral fetch_xor(integral, memory_order = memory_order::seq_cst) noexcept;

integral operator++(int) volatile noexcept;
integral operator++(int) noexcept;
integral operator--(int) volatile noexcept;
integral operator--(int) noexcept;
integral operator++() volatile noexcept;
integral operator++() noexcept;
integral operator--() volatile noexcept;
integral operator--() noexcept;
integral operator+=(integral) volatile noexcept;
integral operator+=(integral) noexception;
integral operator-=(integral) volatile noexcept;
integral operator-=(integral) noexception;
integral operator&=(integral) volatile noexcept;
integral operator&=(integral) noexception;
integral operator|=(integral) volatile noexcept;
integral operator|=(integral) noexception;
integral operator^=(integral) volatile noexcept;
integral operator^=(integral) noexception;

void wait(integral, memory_order = memory_order::seq_cst) const volatile noexcept;
void wait(integral, memory_order = memory_order::seq_cst) const noexcept;
void notify_one() volatile noexcept;
void notify_one() noexception;
void notify_all() volatile noexcept;
void notify_all() noexception;

2 The atomic integral specializations are standard-layout structs. They each have a trivial destructor.

3 Descriptions are provided below only for members that differ from the primary template.

4 The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

Table 144: Atomic arithmetic computations

<table>
<thead>
<tr>
<th>key</th>
<th>Op</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td>bitwise inclusive or</td>
</tr>
<tr>
<td>and</td>
<td>&amp;</td>
<td>bitwise and</td>
</tr>
<tr>
<td>sub</td>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>xor</td>
<td>^</td>
<td>bitwise exclusive or</td>
</tr>
</tbody>
</table>

Note 2: There are no undefined results arising from the computation. — end note
T operator op=(T operand) volatile noexcept;
T operator op=(T operand) noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.
Effects: Equivalent to: return fetch_key(operand) op operand;

31.8.4 Specializations for floating-point types [atomics.types.float]

There are specializations of the atomic class template for the floating-point types float, double, and long double. For each such type floating-point, the specialization atomic<floating-point> provides additional atomic operations appropriate to floating-point types.

namespace std {
    template<> struct atomic<float> {
        using value_type = float;
        using difference_type = value_type;

        static constexpr bool is_always_lock_free = implementation-defined;
        bool is_lock_free() const volatile noexcept;
        bool is_lock_free() const noexcept;
        constexpr atomic() noexcept;
        constexpr atomic(float) noexcept;
        atomic(const atomic&) = delete;
        atomic& operator=(const atomic&) = delete;
        atomic& operator=(const atomic&) volatile = delete;

        void store(float, memory_order = memory_order::seq_cst) volatile noexcept;
        void store(float, memory_order = memory_order::seq_cst) noexcept;
        float operator=(float) volatile noexcept;
        float operator=(float) noexcept;
        float load(memory_order = memory_order::seq_cst) volatile noexcept;
        float load(memory_order = memory_order::seq_cst) noexcept;
        operator float() volatile noexcept;
        operator float() noexcept;
        float exchange(float,
                       memory_order = memory_order::seq_cst) volatile noexcept;
        float exchange(float,
                       memory_order = memory_order::seq_cst) noexcept;
        bool compare_exchange_weak(float&, float,
                                     memory_order, memory_order) volatile noexcept;
        bool compare_exchange_weak(float&, float,
                                     memory_order, memory_order) noexcept;
        bool compare_exchange_strong(float&, float,
                                       memory_order, memory_order) volatile noexcept;
        bool compare_exchange_strong(float&, float,
                                       memory_order, memory_order) noexcept;
        bool compare_exchange_weak(float&, float,
                                     memory_order = memory_order::seq_cst) volatile noexcept;
        bool compare_exchange_weak(float&, float,
                                     memory_order = memory_order::seq_cst) noexcept;
        bool compare_exchange_strong(float&, float,
                                       memory_order = memory_order::seq_cst) volatile noexcept;
        bool compare_exchange_strong(float&, float,
                                       memory_order = memory_order::seq_cst) noexcept;

        float fetch_add(float,
                        memory_order = memory_order::seq_cst) volatile noexcept;
        float fetch_add(float,
                        memory_order = memory_order::seq_cst) noexcept;
        float fetch_sub(float,
                        memory_order = memory_order::seq_cst) volatile noexcept;
        float fetch_sub(float,
                        memory_order = memory_order::seq_cst) noexcept;
    }

§ 31.8.4
The atomic floating-point specializations are standard-layout structs. They each have a trivial destructor.

Descriptions are provided below only for members that differ from the primary template.

The following operations perform arithmetic addition and subtraction computations. The key, operator, and computation correspondence are identified in Table 144.

```cpp
T fetch_key(T operand, memory_order order = memory_order::seq_cst) volatile noexcept;
T fetch_key(T operand, memory_order order = memory_order::seq_cst) noexcept;
```

**Constraints:** For the volatile overload of this function, `is_always_lock_free` is true.

**Effects:** Atomically replaces the value pointed to by `this` with the result of the computation applied to the value pointed to by `this` and the given `operand`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (6.9.2).

**Returns:** Atomically, the value pointed to by `this` immediately before the effects.

**Remarks:** If the result is not a representable value for its type (7.1) the result is unspecified, but the operations otherwise have no undefined behavior. Atomic arithmetic operations on `floating-point` should conform to the `std::numeric_limits<floating-point>` traits associated with the floating-point type (17.3.3). The floating-point environment (26.3) for atomic arithmetic operations on `floating-point` may be different than the calling thread’s floating-point environment.

```cpp
T operator+=(T operand) volatile noexcept;
T operator+=(T operand) noexcept;
```

**Constraints:** For the volatile overload of this function, `is_always_lock_free` is true.

**Effects:** Equivalent to: `return fetch_key(operand) op operand;`

**Remarks:** If the result is not a representable value for its type (7.1) the result is unspecified, but the operations otherwise have no undefined behavior. Atomic arithmetic operations on `floating-point` should conform to the `std::numeric_limits<floating-point>` traits associated with the floating-point type (17.3.3). The floating-point environment (26.3) for atomic arithmetic operations on `floating-point` may be different than the calling thread’s floating-point environment.

### 31.8.5 Partial specialization for pointers

```cpp
namespace std {
    template<class T> struct atomic<T*> {
        using value_type = T*;
        using difference_type = ptrdiff_t;

        static constexpr bool is_always_lock_free = implementation-defined;
        bool is_lock_free() const volatile noexcept;
        bool is_lock_free() const noexcept;

        constexpr atomic() noexcept;
        constexpr atomic(T*) noexcept;
        atomic(const atomic&) = delete;
        atomic& operator=(const atomic&) = delete;
        atomic& operator=(const atomic) volatile = delete;
    };
}
```
There is a partial specialization of the `atomic` class template for pointers. Specializations of this partial specialization are standard-layout structs. They each have a trivial destructor.

Descriptions are provided below only for members that differ from the primary template.

The following operations perform pointer arithmetic. The key, operator, and computation correspondence is:

<table>
<thead>
<tr>
<th>Key</th>
<th>Op</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>sub</td>
<td>-</td>
<td>subtraction</td>
</tr>
</tbody>
</table>

T* fetch_key(ptrdiff_t operand, memory_order_order = memory_order::seq_cst) volatile noexcept;
T* fetch_key(ptrdiff_t operand, memory_order order = memory_order::seq_cst) noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Mandates: T is a complete object type.

[Note 1: Pointer arithmetic on void* or function pointers is ill-formed. — end note]

Effects: Atomically replaces the value pointed to by this with the result of the computation applied to the value pointed to by this and the given operand. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (6.9.2).

Returns: Atomically, the value pointed to by this immediately before the effects.

Remarks: The result may be an undefined address, but the operations otherwise have no undefined behavior.

T* operator op=(ptrdiff_t operand) volatile noexcept;
T* operator op=(ptrdiff_t operand) noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Effects: Equivalent to: return fetch_key(operand) op operand;

31.8.6 Member operators common to integers and pointers to objects
[atomics.types.memop]

value_type operator++(int) volatile noexcept;
value_type operator++(int) noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Effects: Equivalent to: return fetch_add(1);

value_type operator--(int) volatile noexcept;
value_type operator--(int) noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Effects: Equivalent to: return fetch_sub(1);

value_type operator++() volatile noexcept;
value_type operator++() noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Effects: Equivalent to: return fetch_add(1) + 1;

value_type operator--() volatile noexcept;
value_type operator--() noexcept;

Constraints: For the volatile overload of this function, is_always_lock_free is true.

Effects: Equivalent to: return fetch_sub(1) - 1;

31.8.7 Partial specializations for smart pointers
[util.smartptr.atomic]

31.8.7.1 General
[util.smartptr.atomic.general]

The library provides partial specializations of the atomic template for shared-ownership smart pointers (20.11). The behavior of all operations is as specified in 31.8, unless specified otherwise. The template parameter T of these partial specializations may be an incomplete type.

All changes to an atomic smart pointer in 31.8.7, and all associated use_count increments, are guaranteed to be performed atomically. Associated use_count decrements are sequenced after the atomic operation, but are not required to be part of it. Any associated deletion and deallocation are sequenced after the atomic update step and are not part of the atomic operation.

[Note 1: If the atomic operation uses locks, locks acquired by the implementation will be held when any use_count adjustments are performed, and will not be held when any destruction or deallocation resulting from this is performed. — end note]

[Example 1:
template<typename T> class atomic_list {
    struct node {
        T t;
        shared_ptr<node> next;
    }
    atomic<shared_ptr<node>> head;

public:
    auto find(T t) const {
        auto p = head.load();
        while (p && p->t != t)
            p = p->next;

        return shared_ptr<node>(move(p));
    }

    void push_front(T t) {
        auto p = make_shared<node>();
        p->t = t;
        p->next = head;
        while (!head.compare_exchange_weak(p->next, p)) {}
    }
};

31.8.7.2 Partial specialization for shared_ptr

namespace std {
    template<class T> struct atomic<shared_ptr<T>> {
        using value_type = shared_ptr<T>;
        static constexpr bool is_always_lock_free = implementation-defined;
        bool is_lock_free() const noexcept;
        constexpr atomic() noexcept;
        atomic(shared_ptr<T> desired) noexcept;
        atomic(const atomic&) = delete;
        void operator=(const atomic&) = delete;
        shared_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
        operator shared_ptr<T>() const noexcept;
        void store(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
        void operator=(shared_ptr<T> desired) noexcept;
        shared_ptr<T> exchange(shared_ptr<T> desired,
                               memory_order order = memory_order::seq_cst) noexcept;
        bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
                                     memory_order success, memory_order failure) noexcept;
        bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
                                      memory_order success, memory_order failure) noexcept;
        bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
                                     memory_order order = memory_order::seq_cst) noexcept;
        bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
                                      memory_order order = memory_order::seq_cst) noexcept;
        void wait(shared_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
        void notify_one() noexcept;
        void notify_all() noexcept;
    }
};
constexpr atomic() noexcept;

atomic(shared_ptr<T> desired) noexcept;

Effects: Initializes the object with the value desired. Initialization is not an atomic operation (6.9.2).

[Note 1: It is possible to have an access to an atomic object A race with its construction, for example, by communicating the address of the just-constructed object A to another thread via memory_order::relaxed operations on a suitable atomic pointer variable, and then immediately accessing A in the receiving thread. This results in undefined behavior. — end note]

void store(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;

Preconditions: order is neither memory_order::consume, memory_order::acquire, nor memory_order::acq_rel.

Effects: Atomically replaces the value pointed to by this with the value of desired as if by p.swap(desired). Memory is affected according to the value of order.

void operator=(shared_ptr<T> desired) noexcept;

Effects: Equivalent to store(desired).

shared_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;

Preconditions: order is neither memory_order::release nor memory_order::acq_rel.

Effects: Memory is affected according to the value of order.

Returns: Atomically returns p.

operator shared_ptr<T>() const noexcept;

Effects: Equivalent to: return load();

shared_ptr<T> exchange(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;

Effects: Atomically replaces p with desired as if by p.swap(desired). Memory is affected according to the value of order. This is an atomic read-modify-write operation (6.9.2.2).

Returns: Atomically returns the value of p immediately before the effects.

bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
memory_order success, memory_order failure) noexcept;

bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
memory_order success, memory_order failure) noexcept;

Preconditions: failure is neither memory_order::release nor memory_order::acq_rel.

Effects: If p is equivalent to expected, assigns desired to p and has synchronization semantics corresponding to the value of success, otherwise assigns p to expected and has synchronization semantics corresponding to the value of failure.

Returns: true if p was equivalent to expected, false otherwise.

Remarks: Two shared_ptr objects are equivalent if they store the same pointer value and either share ownership or are both empty. The weak form may fail spuriously. See 31.8.2.

If the operation returns true, expected is not accessed after the atomic update and the operation is an atomic read-modify-write operation (6.9.2) on the memory pointed to by this. Otherwise, the operation is an atomic load operation on that memory, and expected is updated with the existing value read from the atomic object in the attempted atomic update. The use_count update corresponding to the write to expected is part of the atomic operation. The write to expected itself is not required to be part of the atomic operation.

bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
memory_order order = memory_order::seq_cst) noexcept;

Effects: Equivalent to:

return compare_exchange_weak(expected, desired, order, fail_order);
where \texttt{fail\_order} is the same as \texttt{order} except that a value of \texttt{memory\_order::acq\_rel} shall be replaced by the value \texttt{memory\_order::acquire} and a value of \texttt{memory\_order::release} shall be replaced by the value \texttt{memory\_order::relaxed}.

\begin{verbatim}
bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
\end{verbatim}

\textit{Effects}: Equivalent to:

\begin{verbatim}
return compare_exchange_strong(expected, desired, order, fail_order);
\end{verbatim}

where \texttt{fail\_order} is the same as \texttt{order} except that a value of \texttt{memory\_order::acq\_rel} shall be replaced by the value \texttt{memory\_order::acquire} and a value of \texttt{memory\_order::release} shall be replaced by the value \texttt{memory\_order::relaxed}.

\begin{verbatim}
void wait(shared_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
\end{verbatim}

\textit{Preconditions}: \texttt{order} is neither \texttt{memory\_order::release} nor \texttt{memory\_order::acq\_rel}.

\textit{Effects}: Repeatedly performs the following steps, in order:

\begin{enumerate}
\item Evaluates \texttt{load(order)} and compares it to \texttt{old}.
\item If the two are not equivalent, returns.
\item Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.
\end{enumerate}

\textit{Remarks}: Two \texttt{shared\_ptr} objects are equivalent if they store the same pointer and either share ownership or are both empty. This function is an atomic waiting operation (31.6).

\begin{verbatim}
void notify_one() noexcept;
\end{verbatim}

\textit{Effects}: Unblocks the execution of at least one atomic waiting operation that is eligible to be unblocked (31.6) by this call, if any such atomic waiting operations exist.

\textit{Remarks}: This function is an atomic notifying operation (31.6).

\begin{verbatim}
void notify_all() noexcept;
\end{verbatim}

\textit{Effects}: Unblocks the execution of all atomic waiting operations that are eligible to be unblocked (31.6) by this call.

\textit{Remarks}: This function is an atomic notifying operation (31.6).

### 31.8.7.3 Partial specialization for weak\_ptr

\begin{verbatim}
namespace std {
    template<class T> struct atomic<weak_ptr<T>> {
        using value_type = weak_ptr<T>;

        static constexpr bool is_always_lock_free = implementation-defined;
        bool is_lock_free() const noexcept;

        constexpr atomic() noexcept;
        atomic(weak_ptr<T> desired) noexcept;
        atomic(const atomic&) = delete;
        void operator=(const atomic&) = delete;

        weak_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
        operator weak_ptr<T>() const noexcept;
        void store(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
        void operator=(weak_ptr<T> desired) noexcept;

        weak_ptr<T> exchange(weak_ptr<T>& desired,
            memory_order order = memory_order::seq_cst) noexcept;
        bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
            memory_order success, memory_order failure) noexcept;
        bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
            memory_order success, memory_order failure) noexcept;
        bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
            memory_order order = memory_order::seq_cst) noexcept;
    }
}
\end{verbatim}
bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired, 
  memory_order order = memory_order::seq_cst) noexcept;

void wait(weak_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
void notify_one() noexcept;
void notify_all() noexcept;

private:
  weak_ptr<T> p;  // exposition only
};

constexpr atomic() noexcept;

Effects: Initializes p{}.
atomic(weak_ptr<T> desired) noexcept;
Effects: Initializes the object with the value desired. Initialization is not an atomic operation (6.9.2).
[Note 1: It is possible to have an access to an atomic object $A$ race with its construction, for example, by communicating the address of the just-constructed object $A$ to another thread via memory_order::relaxed operations on a suitable atomic pointer variable, and then immediately accessing $A$ in the receiving thread. This results in undefined behavior. — end note]

void store(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
Preconditions: order is neither memory_order::consume, memory_order::acquire, nor memory_order::acq_rel.
Effects: Atomically replaces the value pointed to by this with the value of desired as if by p.swap(desired). Memory is affected according to the value of order.

void operator=(weak_ptr<T> desired) noexcept;
Effects: Equivalent to store(desired).

weak_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
Preconditions: order is neither memory_order::release nor memory_order::acq_rel.
Effects: Memory is affected according to the value of order.
Returns: Atomically returns p.

operator weak_ptr<T>() const noexcept;
Effects: Equivalent to: return load();

weak_ptr<T> exchange(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
Effects: Atomically replaces p with desired as if by p.swap(desired). Memory is affected according to the value of order. This is an atomic read-modify-write operation (6.9.2.2).
Returns: Atomically returns the value of p immediately before the effects.

bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired, 
  memory_order success, memory_order failure) noexcept;

bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired, 
  memory_order success, memory_order failure) noexcept;

Preconditions: failure is neither memory_order::release nor memory_order::acq_rel.
Effects: If p is equivalent to expected, assigns desired to p and has synchronization semantics corresponding to the value of success, otherwise assigns p to expected and has synchronization semantics corresponding to the value of failure.
Returns: true if p was equivalent to expected, false otherwise.
Remarks: Two weak_ptr objects are equivalent if they store the same pointer value and either share ownership or are both empty. The weak form may fail spuriously. See 31.8.2.

If the operation returns true, expected is not accessed after the atomic update and the operation is an atomic read-modify-write operation (6.9.2) on the memory pointed to by this. Otherwise, the
operation is an atomic load operation on that memory, and \texttt{expected} is updated with the existing value read from the atomic object in the attempted atomic update. The \texttt{use_count} update corresponding to the write to \texttt{expected} is part of the atomic operation. The write to \texttt{expected} itself is not required to be part of the atomic operation.

\begin{verbatim}
bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired, 
memory_order order = memory_order::seq_cst) noexcept;
\end{verbatim}

17 \textit{Effects}: Equivalent to:
\begin{verbatim}
return compare_exchange_weak(expected, desired, order, fail_order);
\end{verbatim}

where \texttt{fail_order} is the same as \texttt{order} except that a value of \texttt{memory_order::acq_rel} shall be replaced by the value \texttt{memory_order::acquire} and a value of \texttt{memory_order::release} shall be replaced by the value \texttt{memory_order::relaxed}.

\begin{verbatim}
bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired, 
memory_order order = memory_order::seq_cst) noexcept;
\end{verbatim}

18 \textit{Effects}: Equivalent to:
\begin{verbatim}
return compare_exchange_strong(expected, desired, order, fail_order);
\end{verbatim}

where \texttt{fail_order} is the same as \texttt{order} except that a value of \texttt{memory_order::acq_rel} shall be replaced by the value \texttt{memory_order::acquire} and a value of \texttt{memory_order::release} shall be replaced by the value \texttt{memory_order::relaxed}.

\begin{verbatim}
void wait(weak_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
\end{verbatim}

19 \textit{Preconditions}: \texttt{order} is neither \texttt{memory_order::release} nor \texttt{memory_order::acq_rel}.

20 \textit{Effects}: Repeatedly performs the following steps, in order:
\begin{enumerate}[(20.1)]
\item Evaluates \texttt{load(order)} and compares it to \texttt{old}.
\item If the two are not equivalent, returns.
\item Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.
\end{enumerate}

21 \textit{Remarks}: Two \texttt{weak_ptr} objects are equivalent if they store the same pointer and either share ownership or are both empty. This function is an atomic waiting operation (31.6).

\begin{verbatim}
void notify_one() noexcept;
\end{verbatim}

22 \textit{Effects}: Unblocks the execution of at least one atomic waiting operation that is eligible to be unblocked (31.6) by this call, if any such atomic waiting operations exist.

23 \textit{Remarks}: This function is an atomic notifying operation (31.6).

\begin{verbatim}
void notify_all() noexcept;
\end{verbatim}

24 \textit{Effects}: Unblocks the execution of all atomic waiting operations that are eligible to be unblocked (31.6) by this call.

25 \textit{Remarks}: This function is an atomic notifying operation (31.6).

### 31.9 Non-member functions

A non-member function template whose name matches the pattern \texttt{atomic\_f} or the pattern \texttt{atomic\_f\_explicit} invokes the member function \texttt{f}, with the value of the first parameter as the object expression and the values of the remaining parameters (if any) as the arguments of the member function call, in order. An argument for a parameter of type \texttt{atomic\_T::value\_type*} is dereferenced when passed to the member function call. If no such member function exists, the program is ill-formed.

\[\text{Note 1}: \text{The non-member functions enable programmers to write code that can be compiled as either C or C++, for example in a shared header file. — end note}\]

### 31.10 Flag type and operations

\begin{verbatim}
namespace std {
struct atomic_flag {
    constexpr atomic_flag() noexcept;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
}\end{verbatim}
The `atomic_flag` type provides the classic test-and-set functionality. It has two states, set and clear. Operations on an object of type `atomic_flag` shall be lock-free. The operations should also be address-free. The `atomic_flag` type is a standard-layout struct. It has a trivial destructor.

```cpp
constexpr atomic_flag::atomic_flag() noexcept;
```

**Effects:** Initializes *this to the clear state.

```cpp
bool atomic_flag_test(const volatile atomic_flag* object) noexcept;
bool atomic_flag_test(const atomic_flag* object) noexcept;
bool atomic_flag_test_explicit(const volatile atomic_flag* object,
                               memory_order order) noexcept;
bool atomic_flag_test_explicit(const atomic_flag* object,
                               memory_order order) noexcept;
bool atomic_flag::test(memory_order order = memory_order::seq_cst) const volatile noexcept;
bool atomic_flag::test(memory_order order = memory_order::seq_cst) const noexcept;
```

For `atomic_flag_test`, let `order` be `memory_order::seq_cst`.

**Preconditions:** `order` is neither `memory_order::release` nor `memory_order::acq_rel`.

**Effects:** Memory is affected according to the value of `order`.

**Returns:** Atomically returns the value pointed to by `object` or `this`.

```cpp
bool atomic_flag_test_and_set(volatile atomic_flag* object) noexcept;
bool atomic_flag_test_and_set(atomic_flag* object) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag* object, memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag* object, memory_order order) noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order::seq_cst) volatile noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order::seq_cst) noexcept;
```

**Effects:** Atomically sets the value pointed to by `object` or by `this` to `true`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (6.9.2).

**Returns:** Atomically, the value of the object immediately before the effects.

```cpp
void atomic_flag_clear(volatile atomic_flag* object) noexcept;
void atomic_flag_clear(atomic_flag* object) noexcept;
void atomic_flag_clear_explicit(volatile atomic_flag* object, memory_order order) noexcept;
void atomic_flag_clear_explicit(atomic_flag* object, memory_order order) noexcept;
void atomic_flag::clear(memory_order order = memory_order::seq_cst) volatile noexcept;
void atomic_flag::clear(memory_order order = memory_order::seq_cst) noexcept;
```

**Preconditions:** The `order` argument is neither `memory_order::consume`, `memory_order::acquire`, nor `memory_order::acq_rel`.

**Effects:** Atomically sets the value pointed to by `object` or by `this` to `false`. Memory is affected according to the value of `order`.

§ 31.10 1568
void atomic_flag_wait(const volatile atomic_flag* object, bool old) noexcept;
void atomic_flag_wait(const atomic_flag* object, bool old) noexcept;
void atomic_flag_wait_explicit(const volatile atomic_flag* object,
     bool old, memory_order order) noexcept;
void atomic_flag_wait_explicit(const atomic_flag* object,
     bool old, memory_order order) noexcept;
void atomic_flag::wait(bool old, memory_order order =
memory_order::seq_cst) const volatile noexcept;
void atomic_flag::wait(bool old, memory_order order =
memory_order::seq_cst) const noexcept;

For atomic_flag_wait, let order be memory_order::seq_cst. Let flag be object for the non-
member functions and this for the member functions.

Preconditions: order is neither memory_order::release nor memory_order::acq_rel.

Effects: Repeatedly performs the following steps, in order:
(15.1) Evaluates flag->test(order) != old.
(15.2) If the result of that evaluation is true, returns.
(15.3) Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.

Remarks: This function is an atomic waiting operation (31.6).

void atomic_flag_notify_one(volatile atomic_flag* object) noexcept;
void atomic_flag_notify_one(atomic_flag* object) noexcept;
void atomic_flag::notify_one() volatile noexcept;
void atomic_flag::notify_one() noexcept;

Effects: Unblocks the execution of at least one atomic waiting operation that is eligible to be unblocked
(31.6) by this call, if any such atomic waiting operations exist.

Remarks: This function is an atomic notifying operation (31.6).

void atomic_flag_notify_all(volatile atomic_flag* object) noexcept;
void atomic_flag_notify_all(atomic_flag* object) noexcept;
void atomic_flag::notify_all() volatile noexcept;
void atomic_flag::notify_all() noexcept;

Effects: Unblocks the execution of all atomic waiting operations that are eligible to be unblocked (31.6)
by this call.

Remarks: This function is an atomic notifying operation (31.6).

31.11 Fences [atomics.fences]

This subclause introduces synchronization primitives called fences. Fences can have acquire semantics, release
semantics, or both. A fence with acquire semantics is called an acquire fence. A fence with release semantics
is called a release fence.

A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both
operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced
before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release
sequence X would head if it were a release operation.

A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic
object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B
reads the value written by X or a value written by any side effect in the hypothetical release sequence X
would head if it were a release operation.

An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence
B if there exists some atomic operation X on M such that X is sequenced before B and reads the value
written by A or a value written by any side effect in the release sequence headed by A.

extern "C" void atomic_thread_fence(memory_order order) noexcept;

Effects: Depending on the value of order, this operation:
(5.1) has no effects, if order == memory_order::relaxed;
(5.2) is an acquire fence, if order == memory_order::acquire or order == memory_order::consume;
is a release fence, if \texttt{order == memory\_order::\texttt{release}};

— is both an acquire fence and a release fence, if \texttt{order == memory\_order::\texttt{acq\_rel}};

— is a sequentially consistent acquire and release fence, if \texttt{order == memory\_order::\texttt{seq\_cst}}.

\begin{verbatim}
extern "C" void atomic_signal_fence(memory_order order) noexcept;
\end{verbatim}

6 Effects: Equivalent to \texttt{atomic\_thread\_fence(order)}, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

7 \begin{itemize}
\item \texttt{atomic\_signal\_fence} can be used to specify the order in which actions performed by the thread become visible to the signal handler. Compiler optimizations and reorderings of loads and stores are inhibited in the same way as with \texttt{atomic\_thread\_fence}, but the hardware fence instructions that \texttt{atomic\_thread\_fence} would have inserted are not emitted. \textit{— end note}\end{itemize}
32 Thread support library

32.1 General

The following subclauses describe components to create and manage threads (6.9.2), perform mutual exclusion, and communicate conditions and values between threads, as summarized in Table 146.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.2 Requirements</td>
<td></td>
</tr>
<tr>
<td>32.3 Stop tokens</td>
<td>&lt;stop_token&gt;</td>
</tr>
<tr>
<td>32.4 Threads</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>32.5 Mutual exclusion</td>
<td>&lt;mutex&gt;, &lt;shared_mutex&gt;</td>
</tr>
<tr>
<td>32.6 Condition variables</td>
<td>&lt;condition_variable&gt;</td>
</tr>
<tr>
<td>32.7 Semaphores</td>
<td>&lt;semaphore&gt;</td>
</tr>
<tr>
<td>32.8 Coordination types</td>
<td>&lt;latch&gt;, &lt;barrier&gt;</td>
</tr>
<tr>
<td>32.9 Futures</td>
<td>&lt;future&gt;</td>
</tr>
</tbody>
</table>

32.2 Requirements

32.2.1 Template parameter names

Throughout this Clause, the names of template parameters are used to express type requirements. If a template parameter is named `Predicate`, `operator()` applied to the template argument shall return a value that is convertible to `bool`. If a template parameter is named `Clock`, the corresponding template argument shall be a type `C` for which `is_clock_v<C>` is `true`; otherwise the program is ill-formed.

32.2.2 Exceptions

Some functions described in this Clause are specified to throw exceptions of type `system_error` (19.5.8). Such exceptions are thrown if any of the function’s error conditions is detected or a call to an operating system or other underlying API results in an error that prevents the library function from meeting its specifications. Failure to allocate storage is reported as described in 16.4.6.13.

[Example 1: Consider a function in this Clause that is specified to throw exceptions of type `system_error` and specifies error conditions that include `operation_not_permitted` for a thread that does not have the privilege to perform the operation. Assume that, during the execution of this function, an `errno` of `EPERM` is reported by a POSIX API call used by the implementation. Since POSIX specifies an `errno` of `EPERM` when “the caller does not have the privilege to perform the operation”, the implementation maps `EPERM` to an `error_condition` of `operation_not_permitted` (19.5) and an exception of type `system_error` is thrown. — end example]

The `error_code` reported by such an exception’s `code()` member function compares equal to one of the conditions specified in the function’s error condition element.

32.2.3 Native handles

Several classes described in this Clause have members `native_handle_type` and `native_handle`. The presence of these members and their semantics is implementation-defined.

[Note 1: These members allow implementations to provide access to implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. — end note]

32.2.4 Timing specifications

Several functions described in this Clause take an argument to specify a timeout. These timeouts are specified as either a `duration` or a `time_point` type as specified in Clause 27.

Implementations necessarily have some delay in returning from a timeout. Any overhead in interrupt response, function return, and scheduling induces a “quality of implementation” delay, expressed as duration $D_i$. Ideally, this delay would be zero. Further, any contention for processor and memory resources induces a “quality of
management” delay, expressed as duration \( D_m \). The delay durations may vary from timeout to timeout, but in all cases shorter is better.

3 The functions whose names end in \_for take an argument that specifies a duration. These functions produce relative timeouts. Implementations should use a steady clock to measure time for these functions.\(^{329}\) Given a duration argument \( D_t \), the real-time duration of the timeout is \( D_t + D_i + D_m \).

4 The functions whose names end in \_until take an argument that specifies a time point. These functions produce absolute timeouts. Implementations should use the clock specified in the time point to measure time for these functions. Given a clock time point argument \( C_t \), the clock time point of the return from timeout should be \( C_t + D_i + D_m \) when the clock is not adjusted during the timeout. If the clock is adjusted to the time \( C_a \) during the timeout, the behavior should be as follows:

\[(4.1) \quad \text{if } C_a > C_t, \text{ the waiting function should wake as soon as possible, i.e., } C_a + D_i + D_m \text{, since the timeout is already satisfied. This specification may result in the total duration of the wait decreasing when measured against a steady clock.} \]

\[(4.2) \quad \text{if } C_a \leq C_t, \text{ the waiting function should not time out until } \text{Clock}::\text{now()} \text{ returns a time } C_n \geq C_t, \text{ i.e., waking at } C_t + D_i + D_m. \]

[Note 1: When the clock is adjusted backwards, this specification can result in the total duration of the wait increasing when measured against a steady clock. When the clock is adjusted forwards, this specification can result in the total duration of the wait decreasing when measured against a steady clock. —end note]

An implementation returns from such a timeout at any point from the time specified above to the time it would return from a steady-clock relative timeout on the difference between \( C_t \) and the time point of the call to the \_until function.

Recommended practice: Implementations should decrease the duration of the wait when the clock is adjusted forwards.

5 [Note 2: If the clock is not synchronized with a steady clock, e.g., a CPU time clock, these timeouts might not provide useful functionality. —end note]

6 The resolution of timing provided by an implementation depends on both operating system and hardware. The finest resolution provided by an implementation is called the native resolution.

7 Implementation-provided clocks that are used for these functions meet the \texttt{Cpp17TrivialClock} requirements (27.3).

8 A function that takes an argument which specifies a timeout will throw if, during its execution, a clock, time point, or time duration throws an exception. Such exceptions are referred to as timeout-related exceptions. [Note 3: Instantiations of clock, time point and duration types supplied by the implementation as specified in 27.7 do not throw exceptions. —end note]

### 32.2.5 Requirements for \texttt{Cpp17Lockable} types

#### 32.2.5.1 In general

1 An execution agent is an entity such as a thread that may perform work in parallel with other execution agents.

   [Note 1: Implementations or users can introduce other kinds of agents such as processes or thread-pool tasks. —end note]

The calling agent is determined by context, e.g., the calling thread that contains the call, and so on.

2 [Note 2: Some lockable objects are “agent oblivious” in that they work for any execution agent model because they do not determine or store the agent’s ID (e.g., an ordinary spin lock). —end note]

3 The standard library templates \texttt{unique\_lock} (32.5.5.4), \texttt{shared\_lock} (32.5.5.5), \texttt{scoped\_lock} (32.5.5.3), \texttt{lock\_guard} (32.5.5.2), \texttt{lock, try\_lock} (32.5.6), and \texttt{condition\_variable\_any} (32.6.5) all operate on user-supplied lockable objects. The \texttt{Cpp17BasicLockable} requirements, the \texttt{Cpp17Lockable} requirements, and the \texttt{Cpp17TimedLockable} requirements list the requirements imposed by these library types in order to acquire or release ownership of a lock by a given execution agent.

   [Note 3: The nature of any lock ownership and any synchronization it entails are not part of these requirements. —end note]

\(^{329}\) Implementations for which standard time units are meaningful will typically have a steady clock within their hardware implementation.
32.2.5.2 *Cpp17BasicLockable* requirements

A type \( L \) meets the *Cpp17BasicLockable* requirements if the following expressions are well-formed and have the specified semantics (\( m \) denotes a value of type \( L \)).

\[
\begin{align*}
\text{m.lock()} & \\
\text{Effects:} & \text{Blocks until a lock can be acquired for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.}
\end{align*}
\]

\[
\begin{align*}
\text{m.unlock()} & \\
\text{Preconditions:} & \text{The current execution agent holds a lock on } m. \\
\text{Effects:} & \text{Releases a lock on } m \text{ held by the current execution agent.} \\
\text{Throws:} & \text{Nothing.}
\end{align*}
\]

32.2.5.3 *Cpp17Lockable* requirements

A type \( L \) meets the *Cpp17Lockable* requirements if it meets the *Cpp17BasicLockable* requirements and the following expressions are well-formed and have the specified semantics (\( m \) denotes a value of type \( L \)).

\[
\begin{align*}
\text{m.try_lock()} & \\
\text{Effects:} & \text{Attempts to acquire a lock for the current execution agent without blocking. If an exception is thrown then a lock shall not have been acquired for the current execution agent.} \\
\text{Return type:} & \text{bool.} \\
\text{Returns:} & \text{true if the lock was acquired, otherwise } false.
\end{align*}
\]

32.2.5.4 *Cpp17TimedLockable* requirements

A type \( L \) meets the *Cpp17TimedLockable* requirements if it meets the *Cpp17Lockable* requirements and the following expressions are well-formed and have the specified semantics (\( m \) denotes a value of type \( L \), \( \text{rel.time} \) denotes a value of an instantiation of \( \text{duration} \) (27.5), and \( \text{abs.time} \) denotes a value of an instantiation of \( \text{time_point} \) (27.6)).

\[
\begin{align*}
\text{m.try_lock_for(rel_time)} & \\
\text{Effects:} & \text{Attempts to acquire a lock for the current execution agent within the relative timeout (32.2.4) specified by } \text{rel.time}. \text{ The function will not return within the timeout specified by } \text{rel.time} \text{ unless it has obtained a lock on } m \text{ for the current execution agent. If an exception is thrown then a lock has not been acquired for the current execution agent.} \\
\text{Return type:} & \text{bool.} \\
\text{Returns:} & \text{true if the lock was acquired, otherwise } false.
\end{align*}
\]

\[
\begin{align*}
\text{m.try_lock_until(abs_time)} & \\
\text{Effects:} & \text{Attempts to acquire a lock for the current execution agent before the absolute timeout (32.2.4) specified by } \text{abs.time}. \text{ The function will not return before the timeout specified by } \text{abs.time} \text{ unless it has obtained a lock on } m \text{ for the current execution agent. If an exception is thrown then a lock has not been acquired for the current execution agent.} \\
\text{Return type:} & \text{bool.} \\
\text{Returns:} & \text{true if the lock was acquired, otherwise } false.
\end{align*}
\]

32.3 Stop tokens

32.3.1 Introduction

Subclause 32.3 describes components that can be used to asynchronously request that an operation stops execution in a timely manner, typically because the result is no longer required. Such a request is called a *stop request*.

\[
\begin{align*}
\text{stop_source}, \text{stop_token}, \text{and stop_callback} \text{ implement semantics of shared ownership of a } \text{stop state}. \text{ Any } \text{stop_source}, \text{stop_token}, \text{or stop_callback} \text{ that shares ownership of the same stop state is an associated } \text{stop_source}, \text{stop_token}, \text{or stop_callback}, \text{ respectively. The last remaining owner of the stop state automatically releases the resources associated with the stop state.}
\end{align*}
\]
A `stop_token` can be passed to an operation which can either

(3.1) — actively poll the token to check if there has been a stop request, or

(3.2) — register a callback using the `stop_callback` class template which will be called in the event that a stop request is made.

A stop request made via a `stop_source` will be visible to all associated `stop_token` and `stop_source` objects. Once a stop request has been made it cannot be withdrawn (a subsequent stop request has no effect).

Calls to the functions `request_stop`, `stop_requested`, and `stop_possible` do not introduce data races. A call to `request_stop` that returns `true` synchronizes with a call to `stop_requested` on an associated `stop_token` or `stop_source` object that returns `true`. Registration of a callback synchronizes with the invocation of that callback.

### 32.3.2 Header `<stop_token>` synopsis

```cpp
namespace std {
    // 32.3.3, class stop_token
    class stop_token;

    // 32.3.4, class stop_source
    class stop_source;

    // no-shared-stop-state indicator
    struct nostopstate_t {
        explicit nostopstate_t() = default;
    };
    inline constexpr nostopstate_t nostopstate{};

    // 32.3.5, class stop_callback
    template<class Callback>
    class stop_callback;
}
```

### 32.3.3 Class `stop_token`

#### 32.3.3.1 General

The class `stop_token` provides an interface for querying whether a stop request has been made (`stop_requested`) or can ever be made (`stop_possible`) using an associated `stop_source` object (32.3.4). A `stop_token` can also be passed to a `stop_callback` (32.3.5) constructor to register a callback to be called when a stop request has been made from an associated `stop_source`.

```cpp
namespace std {
    class stop_token {
        public:
            // 32.3.3.2, constructors, copy, and assignment
            stop_token() noexcept;
            stop_token(const stop_token&) noexcept;
            stop_token(stop_token&&) noexcept;
            stop_token& operator=(const stop_token&) noexcept;
            stop_token& operator=(stop_token&&) noexcept;
            ~stop_token();
            void swap(stop_token&) noexcept;

            // 32.3.3.3, stop handling
            [[nodiscard]] bool stop_requested() const noexcept;
            [[nodiscard]] bool stop_possible() const noexcept;

            [[nodiscard]] friend bool operator==(const stop_token& lhs, const stop_token& rhs) noexcept;
            friend void swap(stop_token& lhs, stop_token& rhs) noexcept;
    };
}
```
32.3.3.2 Constructors, copy, and assignment

stop_token() noexcept;

Postconditions: stop_possible() is false and stop_requested() is false.

[Note 1: Because the created stop_token object can never receive a stop request, no resources are allocated for a stop state. — end note]

stop_token(const stop_token& rhs) noexcept;

Postconditions: *this == rhs is true.

[Note 2: *this and rhs share the ownership of the same stop state, if any. — end note]

stop_token(stop_token&& rhs) noexcept;

Postconditions: *this contains the value of rhs prior to the start of construction and rhs.stop_possible() is false.

~stop_token();

Effects: Releases ownership of the stop state, if any.

stop_token& operator=(const stop_token& rhs) noexcept;

Effects: Equivalent to: stop_token(rhs).swap(*this).

Returns: *this.

stop_token& operator=(stop_token&& rhs) noexcept;

Effects: Equivalent to: stop_token(std::move(rhs)).swap(*this).

Returns: *this.

void swap(stop_token& rhs) noexcept;

Effects: Exchanges the values of *this and rhs.

32.3.3.3 Members

[[nodiscard]] bool stop_requested() const noexcept;

Returns: true if *this has ownership of a stop state that has received a stop request; otherwise, false.

[[nodiscard]] bool stop_possible() const noexcept;

Returns: false if:

(2.1) *this does not have ownership of a stop state, or

(2.2) a stop request was not made and there are no associated stop_source objects; otherwise, true.

32.3.3.4 Non-member functions

[[nodiscard]] bool operator==(const stop_token& lhs, const stop_token& rhs) noexcept;

Returns: true if lhs and rhs have ownership of the same stop state or if both lhs and rhs do not have ownership of a stop state; otherwise false.

friend void swap(stop_token& x, stop_token& y) noexcept;

Effects: Equivalent to: x.swap(y).

32.3.4 Class stop_source

32.3.4.1 General

The class stop_source implements the semantics of making a stop request. A stop request made on a stop_source object is visible to all associated stop_source and stop_token (32.3.3) objects. Once a stop request has been made it cannot be withdrawn (a subsequent stop request has no effect).
namespace std {

// no-shared-stop-state indicator
struct nostopstate_t {
    explicit nostopstate_t() = default;
};
inline constexpr nostopstate_t nostopstate{};

class stop_source { // 32.3.4.2, constructors, copy, and assignment
public:
    stop_source();
    explicit stop_source(nostopstate_t) noexcept;
    stop_source(const stop_source&) noexcept;
    stop_source(stop_source&&) noexcept;
    stop_source& operator=(const stop_source&) noexcept;
    stop_source& operator=(stop_source&&) noexcept;
    ~stop_source();
    void swap(stop_source&) noexcept;

    // 32.3.4.3, stop handling
    [[nodiscard]] stop_token get_token() const noexcept;
    [[nodiscard]] bool stop_possible() const noexcept;
    [[nodiscard]] bool stop_requested() const noexcept;
    bool request_stop() noexcept;

    [[nodiscard]] friend bool operator==(const stop_source& lhs, const stop_source& rhs) noexcept;
    friend void swap(stop_source& lhs, stop_source& rhs) noexcept;
};
}

32.3.4.2 Constructors, copy, and assignment

1. Effects: Initialises *this to have ownership of a new stop state.
2. Postconditions: stop_possible() is true and stop_requested() is false.
3. Throws: bad_alloc if memory could not be allocated for the stop state.

explicit stop_source(nostopstate_t) noexcept;

4. Postconditions: stop_possible() is false and stop_requested() is false.
5. [Note 1: No resources are allocated for the state. — end note]

stop_source(const stop_source& rhs) noexcept;
6. Postconditions: *this == rhs is true.
7. [Note 2: *this and rhs share the ownership of the same stop state, if any. — end note]

stop_source(stop_source&& rhs) noexcept;
8. Postconditions: *this contains the value of rhs prior to the start of construction and rhs.stop_possible() is false.

~stop_source();
9. Effects: Releases ownership of the stop state, if any.

stop_source& operator=(const stop_source& rhs) noexcept;

Returns: *this.

stop_source& operator=(stop_source&& rhs) noexcept;
11. Effects: Equivalent to: stop_source(std::move(rhs)).swap(*this).

§ 32.3.4.2
void swap(stop_source& rhs) noexcept;

Effects: Exchanges the values of \*this and rhs.

32.3.4.3 Members

[[nodiscard]] stop_token get_token() const noexcept;

Returns: stop_token() if stop_possible() is false; otherwise a new associated stop_token object.

[[nodiscard]] bool stop_possible() const noexcept;

Returns: true if \*this has ownership of a stop state; otherwise, false.

[[nodiscard]] bool stop_requested() const noexcept;

Returns: true if \*this has ownership of a stop state that has received a stop request; otherwise, false.

Effects: If \*this does not have ownership of a stop state, returns false. Otherwise, atomically determines whether the owned stop state has received a stop request, and if not, makes a stop request. The determination and making of the stop request are an atomic read-modify-write operation (6.9.2.2). If the request was made, the callbacks registered by associated stop_callback objects are synchronously called. If an invocation of a callback exits via an exception then terminate is called (14.6.2).

[Note 1: A stop request includes notifying all condition variables of type condition_variable_any temporarily registered during an interruptible wait (32.6.5.3). — end note]

Postconditions: stop_possible() is false or stop_requested() is true.

Returns: true if this call made a stop request; otherwise false.

32.3.4.4 Non-member functions

[[nodiscard]] friend bool

operator==(const stop_source& lhs, const stop_source& rhs) noexcept;

Returns: true if lhs and rhs have ownership of the same stop state or if both lhs and rhs do not have ownership of a stop state; otherwise false.

friend void swap(stop_source& x, stop_source& y) noexcept;

Effects: Equivalent to: x.swap(y).

32.3.5 Class template stop_callback

32.3.5.1 General

namespace std {

class stop_callback {

public:

using callback_type = Callback;

// \ref 32.3.5.2, constructors and destructor

template<class C>

explicit stop_callback(const stop_token& st, C& cb)

noexcept(is_nothrow_constructible_v<Callback, C>);

template<class C>

explicit stop_callback(stop_token&& st, C&& cb)

noexcept(is_nothrow_constructible_v<Callback, C>);

-stop_callback();

stop_callback(const stop_callback&) = delete;
stop_callback(stop_callback&&) = delete;
stop_callback& operator=(const stop_callback&) = delete;
stop_callback& operator=(stop_callback&&) = delete;

§ 32.3.5.1
private:
    Callback callback; // exposition only
};

template<class Callback>
stop_callback(stop_token, Callback) -> stop_callback<Callback>;
}

2 **Mandates:** `stop_callback` is instantiated with an argument for the template parameter `Callback` that satisfies both `invocable` and `destructible`.

3 **Preconditions:** `stop_callback` is instantiated with an argument for the template parameter `Callback` that models both `invocable` and `destructible`.

### 32.3.5.2 Constructors and destructor

> [stopcallback.cons]

```cpp
template<class C>
explicit stop_callback(const stop_token& st, C&& cb)
    noexcept(is_nothrow_constructible_v<Callback, C>);

template<class C>
explicit stop_callback(stop_token&& st, C&& cb)
    noexcept(is_nothrow_constructible_v<Callback, C>);
```

1 **Constraints:** `Callback` and `C` satisfy `constructible_from<Callback, C>`.

2 **Preconditions:** `Callback` and `C` model `constructible_from<Callback, C>`.

3 **Effects:** Initializes `callback` with `std::forward<C>(cb)`. If `st.stop_requested()` is true, then `std::forward<Callback>(callback)()` is evaluated in the current thread before the constructor returns. Otherwise, if `st` has ownership of a stop state, acquires shared ownership of that stop state and registers the callback with that stop state such that `std::forward<Callback>(callback)()` is evaluated by the first call to `request_stop()` on an associated `stop_source`.

4 **Throws:** Any exception thrown by the initialization of `callback`.

5 **Remarks:** If evaluating `std::forward<Callback>(callback)()` exits via an exception, then `terminate` is called (14.6.2).

```cpp
~stop_callback();
```

6 **Effects:** Unregisters the callback from the owned stop state, if any. The destructor does not block waiting for the execution of another callback registered by an associated `stop_callback`. If `callback` is concurrently executing on another thread, then the return from the invocation of `callback` strongly happens before (6.9.2.2) `callback` is destroyed. If `callback` is executing on the current thread, then the destructor does not block (3.7) waiting for the return from the invocation of `callback`. Releases ownership of the stop state, if any.

### 32.4 Threads

> [thread.threads]

### 32.4.1 General

> [thread.threads.general]

1 32.4 describes components that can be used to create and manage threads.

[Note 1: These threads are intended to map one-to-one with operating system threads. — end note]

### 32.4.2 Header `<thread>` synopsis

> [thread.syn]

```cpp
#include <compare> // see 17.11.1

namespace std {
    class thread;

    void swap(thread& x, thread& y) noexcept;

    // 32.4.4 class jthread
    class jthread;

    namespace this_thread {
        thread::id get_id() noexcept;
    }
```
void yield() noexcept;

template<class Clock, class Duration>
void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);

template<class Rep, class Period>
void sleep_for(const chrono::duration<Rep, Period>& rel_time);

} } } 

32.4.3 Class thread

32.4.3.1 General

The class thread provides a mechanism to create a new thread of execution, to join with a thread (i.e., wait for a thread to complete), and to perform other operations that manage and query the state of a thread. A thread object uniquely represents a particular thread of execution. That representation may be transferred to other thread objects in such a way that no two thread objects simultaneously represent the same thread of execution. A thread of execution is detached when no thread object represents that thread. Objects of class thread can be in a state that does not represent a thread of execution.

[Note 1: A thread object does not represent a thread of execution after default construction, after being moved from, or after a successful call to detach or join. — end note]
An object of type `thread::id` provides a unique identifier for each thread of execution and a single distinct value for all `thread` objects that do not represent a thread of execution (32.4.3). Each thread of execution has an associated `thread::id` object that is not equal to the `thread::id` object of any other thread of execution and that is not equal to the `thread::id` object of any `thread` object that does not represent threads of execution.

`thread::id` is a trivially copyable class (11.2). The library may reuse the value of a `thread::id` of a terminated thread that can no longer be joined.

[Note 1: Relational operators allow `thread::id` objects to be used as keys in associative containers. — end note]

```cpp
id() noexcept;
```

**Postconditions:** The constructed object does not represent a thread of execution.

```cpp
bool operator==(thread::id x, thread::id y) noexcept;
```

**Returns:** `true` only if `x` and `y` represent the same thread of execution or neither `x` nor `y` represents a thread of execution.

```cpp
strong_ordering operator<=>(thread::id x, thread::id y) noexcept;
```

Let `P(x,y)` be an unspecified total ordering over `thread::id` as described in 25.8.

**Returns:** `strong_ordering::less` if `P(x,y)` is `true`. Otherwise, `strong_ordering::greater` if `P(y,x)` is `true`. Otherwise, `strong_ordering::equal`.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& out, thread::id id);
```

**Effects:** Inserts an unspecified text representation of `id` into `out`. For two objects of type `thread::id x` and `y`, if `x == y` the `thread::id` objects have the same text representation and if `x != y` the `thread::id` objects have distinct text representations.

**Returns:** `out`.

The specialization is enabled (20.14.19).

### 32.4.3.3 Constructors

**thread() noexcept;**

**Effects:** The object does not represent a thread of execution.

**Postconditions:** `get_id() == id()`.

**template<class F, class... Args> explicit thread(F&& f, Args&&... args);**

**Constraints:** `remove_cvref_t<F>` is not the same type as `thread`.

**Mandates:** The following are all `true`:

(4.1) `is_constructible_v<decay_t<F>, F>`,
(4.2) `is_constructible_v<decay_t<Args>, Args && ...>`,
(4.3) `is_move_constructible_v<decay_t<F>, F>`,
(4.4) `is_move_constructible_v<decay_t<Args>, Args && ...>`, and
(4.5) `is_invocable_v<decay_t<F>, decay_t<Args>...>`.

**Preconditions:** `decay_t<F>` and each type in `decay_t<Args>` meet the `Cpp17MoveConstructible` requirements.

**Effects:** The new thread of execution executes

```cpp
invoke(decay-copy(std::forward<F>(f)), decay-copy(std::forward<Args>(args))...)
```
with the calls to decay-copy being evaluated in the constructing thread. Any return value from this invocation is ignored.

[Note 1: This implies that any exceptions not thrown from the invocation of the copy of \( f \) will be thrown in the constructing thread, not the new thread. — end note]

If the invocation of invoke terminates with an uncaught exception, terminate is called.

Synchronization: The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of \( f \).

Postconditions: get_id() != id(). \*this represents the newly started thread.

Throws: system_error if unable to start the new thread.

Error conditions:
- resource_unavailable_try_again — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

\[
\text{thread(thread&& x) noexcept;}
\]

Postconditions: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the start of construction.

32.4.3.4 Destructor

\[
\sim\text{thread();}
\]

Effects: If joinable(), calls terminate(). Otherwise, has no effects.

[Note 1: Either implicitly detaching or joining a joinable() thread in its destructor could result in difficult to debug correctness (for detach) or performance (for join) bugs encountered only when an exception is thrown. These bugs can be avoided by ensuring that the destructor is never executed while the thread is still joinable. —end note]

32.4.3.5 Assignment

\[
\text{thread& operator=(thread&& x) noexcept;}
\]

Effects: If joinable(), calls terminate(). Otherwise, assigns the state of \( x \) to \*this and sets \( x \) to a default constructed state.

Postconditions: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the assignment.

Returns: \*this.

32.4.3.6 Members

\[
\text{void swap(thread& x) noexcept;}
\]

Effects: Swaps the state of \*this and \( x \).

\[
\text{bool joinable() const noexcept;}
\]

Returns: get_id() != id().

\[
\text{void join();}
\]

Effects: Blocks until the thread represented by \*this has completed.

Synchronization: The completion of the thread represented by \*this synchronizes with (6.9.2) the corresponding successful join() return.

[Note 1: Operations on \*this are not synchronized. — end note]

Postconditions: The thread represented by \*this has completed. get_id() == id().

Throws: system_error when an exception is required (32.2.2).

Error conditions:
- resource_deadlock_would_occur — if deadlock is detected or get_id() == this_thread::get_id().
- no_such_process — if the thread is not valid.
8

Effects: The thread represented by *this continues execution without the calling thread blocking.
When detach() returns, *this no longer represents the possibly continuing thread of execution. When
the thread previously represented by *this ends execution, the implementation releases any owned
resources.

9

Postconditions: get_id() == id().

10

Throws: system_error when an exception is required (32.2.2).

11

Error conditions:

11.1 — no_such_process — if the thread is not valid.

11.2 — invalid_argument — if the thread is not joinable.

12

Returns: A default constructed id object if *this does not represent a thread, otherwise this_-
thread::get_id() for the thread of execution represented by *this.

32.4.3.7 Static members

unsigned hardware_concurrency() noexcept;

Returns: The number of hardware thread contexts.

[Note 1: This value should only be considered to be a hint. — end note]
If this value is not computable or well-defined, an implementation should return 0.

32.4.3.8 Specialized algorithms

void swap(thread& x, thread& y) noexcept;

Effects: As if by x.swap(y).

32.4.4 Class jthread

32.4.4.1 General

The class jthread provides a mechanism to create a new thread of execution. The functionality is the same
as for class thread (32.4.3) with the additional abilities to provide a stop_token (32.3) to the new thread of
execution, make stop requests, and automatically join.
// 32.4.4.4, stop token handling
[[nodiscard]] stop_source get_stop_source() noexcept;
[[nodiscard]] stop_token get_stop_token() const noexcept;
bool request_stop() noexcept;

// 32.4.4.5, specialized algorithms
friend void swap(jthread& lhs, jthread& rhs) noexcept;

// 32.4.4.6, static members
[[nodiscard]] static unsigned int hardware_concurrency() noexcept;

private:
    stop_source ssource;  // exposition only
};

32.4.4.2 Constructors, move, and assignment

jthread() noexcept;
1
Effects: Constructs a jthread object that does not represent a thread of execution.
2
Postconditions: get_id() == id() is true and ssource.stop_possible() is false.
3

template<class F, class... Args> explicit jthread(F&& f, Args&&... args);
4

Constraints: remove_cvref_t<F> is not the same type as jthread.

Mandates: The following are all true:
(4.1) — is_constructible_v<decay_t<F>, F>,
(4.2) — (is_constructible_v<decay_t<Args>, Args> && ...),
(4.3) — is_move_constructible_v<decay_t<F>>,
(4.4) — (is_move_constructible_v<decay_t<Args>> && ...), and
(4.5) — is_invocable_v<decay_t<F>, decay_t<Args>... ||
        is_invocable_v<decay_t<F>, stop_token, decay_t<Args>...>.
5
Preconditions: decay_t<F> and each type in decay_t<Args> meet the Cpp17MoveConstructible requirements.

Effects: Initializes ssource. The new thread of execution executes
        invoke(decay-copy(std::forward<F>(f)), get_stop_token(),
              decay-copy(std::forward<Args>(args))...)
if that expression is well-formed, otherwise
        invoke(decay-copy(std::forward<F>(f)), decay-copy(std::forward<Args>(args))...)
with the calls to decay-copy being evaluated in the constructing thread. Any return value from this
invocation is ignored.

[Note 1: This implies that any exceptions not thrown from the invocation of the copy of f will be thrown in
the constructing thread, not the new thread. — end note]

If the invoke expression exits via an exception, terminate is called.

Synchronization: The completion of the invocation of the constructor synchronizes with the beginning
of the invocation of the copy of f.

Postconditions: get_id() != id() is true and ssource.stop_possible() is true and *this represents
the newly started thread.

[Note 2: The calling thread can make a stop request only once, because it cannot replace this stop token.
— end note]

Throws: system_error if unable to start the new thread.

Error conditions:
(10.1) — resource_unavailable_try_again — the system lacked the necessary resources to create another
thread, or the system-imposed limit on the number of threads in a process would be exceeded.
jthread(jthread&& x) noexcept;

Postconditions: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the start of construction. ssourse has the value of x.ssourse prior to the start of construction and x.ssourse.stop_possible() is false.

-jthread();

Effects: If joinable() is true, calls request_stop() and then join().

[Note 3: Operations on *this are not synchronized. — end note]

jthread& operator=(jthread&& x) noexcept;

Effects: If joinable() is true, calls request_stop() and then join(). Assigns the state of x to *this and sets x to a default constructed state.

Postconditions: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the assignment. ssourse has the value of x.ssourse prior to the assignment and x.ssourse.stop_possible() is false.

Returns: *this.

32.4.4.3 Members

void swap(jthread& x) noexcept;

Effects: Exchanges the values of *this and x.

[[nodiscard]] bool joinable() const noexcept;

Returns: get_id() != id().

void join();

Effects: Blocks until the thread represented by *this has completed.

Synchronization: The completion of the thread represented by *this synchronizes with (6.9.2) the corresponding successful join() return.

[Note 1: Operations on *this are not synchronized. — end note]

Postconditions: The thread represented by *this has completed. get_id() == id().

Throws: system_error when an exception is required (32.2.2).

Error conditions:

(7.1) — resource_deadlock_would_occur — if deadlock is detected or get_id() == this_thread::get_id().

(7.2) — no_such_process — if the thread is not valid.

(7.3) — invalid_argument — if the thread is not joinable.

void detach();

Effects: The thread represented by *this continues execution without the calling thread blocking. When detach() returns, *this no longer represents the possibly continuing thread of execution. When the thread previously represented by *this ends execution, the implementation releases any owned resources.

Postconditions: get_id() == id().

Throws: system_error when an exception is required (32.2.2).

Error conditions:

(11.1) — no_such_process — if the thread is not valid.

(11.2) — invalid_argument — if the thread is not joinable.

id get_id() const noexcept;

Returns: A default constructed id object if *this does not represent a thread, otherwise this_thread::get_id() for the thread of execution represented by *this.
32.4.4.4 Stop token handling
[[nodiscard]] stop_source get_stop_source() noexcept;

Effects: Equivalent to: return ssource;

[[nodiscard]] stop_token get_stop_token() const noexcept;

Effects: Equivalent to: return ssource.get_token();

bool request_stop() noexcept;

Effects: Equivalent to: return ssource.request_stop();

32.4.4.5 Specialized algorithms
friend void swap(jthread& x, jthread& y) noexcept;

Effects: Equivalent to: x.swap(y).

32.4.4.6 Static members
[[nodiscard]] static unsigned int hardware_concurrency() noexcept;

Returns: thread::hardware_concurrency().

32.4.5 Namespace this_thread
namespace std::this_thread {
  thread::id get_id() noexcept;

  void yield() noexcept;
  template<class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
  template<class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
}

thread::id this_thread::get_id() noexcept;

Returns: An object of type thread::id that uniquely identifies the current thread of execution. No other thread of execution has this id and this thread of execution always has this id. The object returned does not compare equal to a default constructed thread::id.

void this_thread::yield() noexcept;

Effects: Offers the implementation the opportunity to reschedule.

Synchronization: None.

template<class Clock, class Duration>
  void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);

Effects: Blocks the calling thread for the absolute timeout (32.2.4) specified by abs_time.

Synchronization: None.

Throws: Timeout-related exceptions (32.2.4).

template<class Rep, class Period>
  void sleep_for(const chrono::duration<Rep, Period>& rel_time);

Effects: Blocks the calling thread for the relative timeout (32.2.4) specified by rel_time.

Synchronization: None.

Throws: Timeout-related exceptions (32.2.4).

32.5 Mutual exclusion

32.5.1 General

Subclause 32.5 provides mechanisms for mutual exclusion: mutexes, locks, and call once. These mechanisms ease the production of race-free programs (6.9.2).
32.5.2 Header `<mutex>` synopsis

```cpp
namespace std {
    class mutex;
    class recursive_mutex;
    class timed_mutex;
    class recursive_timed_mutex;

    struct defer_lock_t { explicit defer_lock_t() = default; };
    struct try_to_lock_t { explicit try_to_lock_t() = default; };
    struct adopt_lock_t { explicit adopt_lock_t() = default; };

    inline constexpr defer_lock_t defer_lock { };
    inline constexpr try_to_lock_t try_to_lock { };
    inline constexpr adopt_lock_t adopt_lock { };

    template<class Mutex> class lock_guard;
    template<class... MutexTypes> class scoped_lock;
    template<class Mutex> class unique_lock;

    template<class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;

    template<class L1, class L2, class... L3>
    int try_lock(L1&, L2&, L3&...);
    template<class L1, class L2, class... L3>
    void lock(L1&, L2&, L3&...);

    struct once_flag;

    template<class Callable, class... Args>
    void call_once(once_flag& flag, Callable&& func, Args&&... args);
}
```

32.5.3 Header `<shared_mutex>` synopsis

```cpp
namespace std {
    class shared_mutex;
    class shared_timed_mutex;
    template<class Mutex> class shared_lock;

    template<class Mutex>
    void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
}
```

32.5.4 Mutex requirements

32.5.4.1 In general

A mutex object facilitates protection against data races and allows safe synchronization of data between execution agents (32.2.5). An execution agent owns a mutex from the time it successfully calls one of the lock functions until it calls unlock. Mutexes can be either recursive or non-recursive, and can grant simultaneous ownership to one or many execution agents. Both recursive and non-recursive mutexes are supplied.

32.5.4.2 Mutex types

The mutex types are the standard library types `mutex`, `recursive_mutex`, `timed_mutex`, `recursive_timed_mutex`, `shared_mutex`, and `shared_timed_mutex`. They meet the requirements set out in 32.5.4.2. In this description, `m` denotes an object of a mutex type.

The mutex types meet the `Cpp17Lockable` requirements (32.2.5.3).

The mutex types meet `Cpp17DefaultConstructible` and `Cpp17Destructible`. If initialization of an object of a mutex type fails, an exception of type `system_error` is thrown. The mutex types are neither copyable nor movable.

The error conditions for error codes, if any, reported by member functions of the mutex types are as follows:

- `resource_unavailable_try_again` — if any native handle type manipulated is not available.
- `operation_not_permitted` — if the thread does not have the privilege to perform the operation.
The implementation provides lock and unlock operations, as described below. For purposes of determining
the existence of a data race, these behave as atomic operations (6.9.2). The lock and unlock operations on a
single mutex appears to occur in a single total order.

[Note 1: This can be viewed as the modification order (6.9.2) of the mutex. — end note]

[Note 2: Construction and destruction of an object of a mutex type need not be thread-safe; other synchronization
can be used to ensure that mutex objects are initialized and visible to other threads. — end note]

The expression `m.lock()` is well-formed and has the following semantics:

**Preconditions:** If `m` is of type `mutex`, `timed_mutex`, `shared_mutex`, or `shared_timed_mutex`, the calling
thread does not own the mutex.

**Effects:** Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.

**Postconditions:** The calling thread owns the mutex.

**Return type:** `void`.

**Synchronization:** Prior `unlock()` operations on the same object synchronize with (6.9.2) this operation.

**Throws:** `system_error` when an exception is required (32.2.2).

**Error conditions:**

- `operation_not_permitted` — if the thread does not have the privilege to perform the operation.
- `resource_deadlock_would_occur` — if the implementation detects that a deadlock would occur.

The expression `m.try_lock()` is well-formed and has the following semantics:

**Preconditions:** If `m` is of type `mutex`, `timed_mutex`, `shared_mutex`, or `shared_timed_mutex`, the calling
thread does not own the mutex.

**Effects:** Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership
is not obtained, there is no effect and `try_lock()` immediately returns. An implementation may fail to
obtain the lock even if it is not held by any other thread.

[Note 3: This spurious failure is normally uncommon, but allows interesting implementations based on a simple
compare and exchange (Clause 31). — end note]

An implementation should ensure that `try_lock()` does not consistently return `false` in the absence
of contending mutex acquisitions.

**Return type:** `bool`.

**Returns:** `true` if ownership was obtained, otherwise `false`.

**Synchronization:** If `try_lock()` returns `true`, prior `unlock()` operations on the same object synchronize
with (6.9.2) this operation.

[Note 4: Since `lock()` does not synchronize with a failed subsequent `try_lock()`, the visibility rules are weak
enough that little would be known about the state after a failure, even in the absence of spurious failures.
— end note]

**Throws:** Nothing.

The expression `m.unlock()` is well-formed and has the following semantics:

**Preconditions:** The calling thread owns the mutex.

**Effects:** Releases the calling thread’s ownership of the mutex.

**Return type:** `void`.

**Synchronization:** This operation synchronizes with (6.9.2) subsequent lock operations that obtain
ownership on the same object.

**Throws:** Nothing.

### 32.5.4.2.2 Class mutex

```cpp
namespace std {
    class mutex {
        public:
            constexpr mutex() noexcept;
            mutex(mutex const &);  // nothrow
            mutex(mutex &&);        // nothrow
            mutex(mutex && other) = delete;  // destroy

            ~mutex();

            mutex& operator=(mutex);  // nothrow
            mutex& operator=(mutex &&);  // nothrow

            bool operator==(mutex) const;  // nothrow
            bool operator!=(mutex) const;  // nothrow

            mutex& lock();
            mutex& try_lock();
            mutex& unlock();

        private:
            friend mutex& operator++(mutex &);
            friend mutex& operator--(mutex &);
    }

    mutex& operator++(mutex &);  // nothrow
    mutex& operator--(mutex &);  // nothrow
}
```
mutex();
mutex(const mutex&) = delete;
mutex& operator=(const mutex&) = delete;

void lock();
bool try_lock();
void unlock();

using native_handle_type = implementation-defined; // see 32.2.3
native_handle_type native_handle(); // see 32.2.3

1 The class mutex provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for try_lock()) or block (for lock()) until the owning thread has released ownership with a call to unlock().

2 [Note 1: After a thread A has called unlock(), releasing a mutex, it is possible for another thread B to lock the same mutex, observe that it is no longer in use, unlock it, and destroy it, before thread A appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread A doesn’t access the mutex after the unlock call returns. These cases typically occur when a reference-counted object contains a mutex that is used to protect the reference count. — end note]

3 The class mutex meets all of the mutex requirements (32.5.4). It is a standard-layout class (11.2).

4 [Note 2: A program can deadlock if the thread that owns a mutex object calls lock() on that object. If the implementation can detect the deadlock, a resource_deadlock_would_occur error condition might be observed. — end note]

5 The behavior of a program is undefined if it destroys a mutex object owned by any thread or a thread terminates while owning a mutex object.

### 32.5.4.2.3 Class recursive_mutex [thread.mutex.recursive]

namespace std {
    class recursive_mutex {
        public:
            recursive_mutex();
            ~recursive_mutex();

            recursive_mutex(const recursive_mutex&) = delete;
            recursive_mutex& operator=(const recursive_mutex&) = delete;

            void lock();
            bool try_lock() noexcept;
            void unlock();

            using native_handle_type = implementation-defined; // see 32.2.3
            native_handle_type native_handle(); // see 32.2.3
    }
}

1 The class recursive_mutex provides a recursive mutex with exclusive ownership semantics. If one thread owns a recursive_mutex object, attempts by another thread to acquire ownership of that object will fail (for try_lock()) or block (for lock()) until the first thread has completely released ownership.

2 The class recursive_mutex meets all of the mutex requirements (32.5.4). It is a standard-layout class (11.2).

3 A thread that owns a recursive_mutex object may acquire additional levels of ownership by calling lock() or try_lock() on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a recursive_mutex object, additional calls to try_lock() fail, and additional calls to lock() throw an exception of type system_error. A thread shall call unlock() once for each level of ownership acquired by calls to lock() and try_lock(). Only when all levels of ownership have been released may ownership be acquired by another thread.

4 The behavior of a program is undefined if:

   (4.1) — it destroys a recursive_mutex object owned by any thread or
— a thread terminates while owning a recursive_mutex object.

32.5.4.3 Timed mutex types

32.5.4.3.1 General

The timed mutex types are the standard library types timed_mutex, recursive_timed_mutex, and shared_timed_mutex. They meet the requirements set out below. In this description, m denotes an object of a mutex type, rel_time denotes an object of an instantiation of duration (27.5), and abs_time denotes an object of an instantiation of time_point (27.6).

The timed mutex types meet the Cpp17TimedLockable requirements (32.2.5.4).

The expression m.try_lock_for(rel_time) is well-formed and has the following semantics:

Preconditions: If m is of type timed_mutex or shared_timed_mutex, the calling thread does not own the mutex.

Effects: The function attempts to obtain ownership of the mutex within the relative timeout (32.2.4) specified by rel_time. If the time specified by rel_time is less than or equal to rel_time.zero(), the function attempts to obtain ownership without blocking (as if by calling try_lock()). The function returns within the timeout specified by rel_time only if it has obtained ownership of the mutex object.

[Note 1: As with try_lock(), there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — end note]

Return type: bool.

Returns: true if ownership was obtained, otherwise false.

Synchronization: If try_lock_for() returns true, prior unlock() operations on the same object synchronize with (6.9.2) this operation.

Throws: Timeout-related exceptions (32.2.4).

The expression m.try_lock_until(abs_time) is well-formed and has the following semantics:

Preconditions: If m is of type timed_mutex or shared_timed_mutex, the calling thread does not own the mutex.

Effects: The function attempts to obtain ownership of the mutex. If abs_time has already passed, the function attempts to obtain ownership without blocking (as if by calling try_lock()). The function returns before the absolute timeout (32.2.4) specified by abs_time only if it has obtained ownership of the mutex object.

[Note 2: As with try_lock(), there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — end note]

Return type: bool.

Returns: true if ownership was obtained, otherwise false.

Synchronization: If try_lock_until() returns true, prior unlock() operations on the same object synchronize with (6.9.2) this operation.

Throws: Timeout-related exceptions (32.2.4).

32.5.4.3.2 Class timed_mutex

namespace std {

    class timed_mutex {
        public:
            timed_mutex();       // blocking
            ~timed_mutex();
            timed_mutex(const timed_mutex&) = delete;
            timed_mutex& operator=(const timed_mutex&) = delete;
            void lock();
            bool try_lock();
            template<typename Rep, typename Period>
                bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
            template<typename Clock, typename Duration>
                bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    }

} // namespace std
The class `timed_mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a `timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`) until the owning thread has released ownership with a call to `unlock()` or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

The class `timed_mutex` meets all of the timed mutex requirements (32.5.4.3). It is a standard-layout class (11.2).

The behavior of a program is undefined if:

- it destroys a `timed_mutex` object owned by any thread,
- a thread that owns a `timed_mutex` object calls `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object, or
- a thread terminates while owning a `timed_mutex` object.

The class `recursive_timed_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`) until the owning thread has completely released ownership or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

The class `recursive_timed_mutex` meets all of the timed mutex requirements (32.5.4.3). It is a standard-layout class (11.2).

A thread that owns a `recursive_timed_mutex` object may acquire additional levels of ownership by calling `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_timed_mutex` object, additional calls to `try_lock()`, `try_lock_for()`, or `try_lock_until()` fail, and additional calls to `lock()` throw an exception of type `system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()`, `try_lock()`, `try_lock_for()`, and `try_lock_until()`. Only when all levels of ownership have been released may ownership of the object be acquired by another thread.

The behavior of a program is undefined if:
— it destroys a recursive_timed_mutex object owned by any thread, or
— a thread terminates while owning a recursive_timed_mutex object.

32.5.4.4 Shared mutex types

32.5.4.4.1 General

The standard library types shared_mutex and shared_timed_mutex are shared mutex types. Shared mutex types meet the requirements of mutex types (32.5.4.2) and additionally meet the requirements set out below. In this description, \( m \) denotes an object of a shared mutex type.

In addition to the exclusive lock ownership mode specified in 32.5.4.2, shared mutex types provide a shared lock ownership mode. Multiple execution agents can simultaneously hold a shared lock ownership of a shared mutex type. But no execution agent holds a shared lock while another execution agent holds an exclusive lock on the same shared mutex type, and vice-versa. The maximum number of execution agents which can share a shared lock on a single shared mutex type is unspecified, but is at least 10000. If more than the maximum number of execution agents attempt to obtain a shared lock, the excess execution agents block until the number of shared locks are reduced below the maximum amount by other execution agents releasing their shared lock.

The expression \( m.lock_shared() \) is well-formed and has the following semantics:

1. **Preconditions**: The calling thread has no ownership of the mutex.
2. **Effects**: Blocks the calling thread until shared ownership of the mutex can be obtained for the calling thread. If an exception is thrown then a shared lock has not been acquired for the current thread.
3. **Postconditions**: The calling thread has a shared lock on the mutex.
4. **Return type**: void.
5. **Synchronization**: Prior unlock() operations on the same object synchronize with (6.9.2) this operation.
6. **Throws**: system_error when an exception is required (32.2.2).
7. **Error conditions**:
   - operation_not_permitted — if the thread does not have the privilege to perform the operation.
   - resource_deadlock_would_occur — if the implementation detects that a deadlock would occur.

The expression \( m.unlock_shared() \) is well-formed and has the following semantics:

1. **Preconditions**: The calling thread holds a shared lock on the mutex.
2. **Effects**: Releases a shared lock on the mutex held by the calling thread.
3. **Return type**: void.
4. **Synchronization**: This operation synchronizes with (6.9.2) subsequent lock() operations that obtain ownership on the same object.
5. **Throws**: Nothing.

The expression \( m.try_lock_shared() \) is well-formed and has the following semantics:

1. **Preconditions**: The calling thread has no ownership of the mutex.
2. **Effects**: Attempts to obtain shared ownership of the mutex for the calling thread without blocking. If shared ownership is not obtained, there is no effect and try_lock_shared() immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread.
3. **Return type**: bool.
4. **Returns**: true if the shared lock was acquired, otherwise false.
5. **Synchronization**: If try_lock_shared() returns true, prior unlock() operations on the same object synchronize with (6.9.2) this operation.
6. **Throws**: Nothing.

32.5.4.4.2 Class shared_mutex

namespace std {
    class shared_mutex {
        public:
            shared_mutex();
    }
}
The class `shared_mutex` provides a non-recursive mutex with shared ownership semantics.

The class `shared_mutex` meets all of the shared mutex requirements (32.5.4.4). It is a standard-layout class (11.2).

The behavior of a program is undefined if:

1. it destroys a `shared_mutex` object owned by any thread,
2. a thread attempts to recursively gain any ownership of a `shared_mutex`, or
3. a thread terminates while possessing any ownership of a `shared_mutex`.

`shared_mutex` may be a synonym for `shared_timed_mutex`.

### 32.5.4.5 Shared timed mutex types

The standard library type `shared_timed_mutex` is a shared timed mutex type. Shared timed mutex types meet the requirements of timed mutex types (32.5.4.3), shared mutex types (32.5.4.4), and additionally meet the requirements set out below. In this description, `m` denotes an object of a shared timed mutex type, `rel_type` denotes an object of an instantiation of `duration` (27.5), and `abs_time` denotes an object of an instantiation of `time_point` (27.6).

The expression `m.try_lock_shared_for(rel_time)` is well-formed and has the following semantics:

- **Preconditions**: The calling thread has no ownership of the mutex.

- **Effects**: Attempts to obtain shared lock ownership for the calling thread within the relative timeout (32.2.4) specified by `rel_time`. If the time specified by `rel_time` is less than or equal to `rel_time.zero()`, the function attempts to obtain ownership without blocking (as if by calling `try_lock_shared()`). The function returns within the timeout specified by `rel_time` only if it has obtained shared ownership of the mutex object.

- **Return type**: `bool`.

- **Returns**: `true` if the shared lock was acquired, otherwise `false`.

- **Synchronization**: If `try_lock_shared_for()` returns `true`, prior `unlock()` operations on the same object synchronize with (6.9.2) this operation.

- **Throws**: Timeout-related exceptions (32.2.4).

The expression `m.try_lock_shared_until(abs_time)` is well-formed and has the following semantics:

- **Preconditions**: The calling thread has no ownership of the mutex.
Effects: The function attempts to obtain shared ownership of the mutex. If `abs_time` has already passed, the function attempts to obtain shared ownership without blocking (as if by calling `try_lock_shared()`). The function returns before the absolute timeout (32.2.4) specified by `abs_time` only if it has obtained shared ownership of the mutex object.

[Note 2: As with `try_lock()`, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — end note]

If an exception is thrown then a shared lock has not been acquired for the current thread.

Return type: `bool`.

Returns: `true` if the shared lock was acquired, otherwise `false`.

Synchronization: If `try_lock_shared_until()` returns `true`, prior `unlock()` operations on the same object synchronize with (6.9.2) this operation.

Throws: Timeout-related exceptions (32.2.4).

32.5.4.5.2 Class `shared_timed_mutex` [thread.sharedtimedmutex.class]

```cpp
namespace std {
    class shared_timed_mutex {
        public:
            shared_timed_mutex();
            ~shared_timed_mutex();
            shared_timed_mutex(const shared_timed_mutex&) = delete;
            shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;

            // exclusive ownership
            void lock();          // blocking
            bool try_lock();
            template<class Rep, class Period>
                bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
            template<class Clock, class Duration>
                bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
            void unlock();

            // shared ownership
            void lock_shared();    // blocking
            bool try_lock_shared();
            template<class Rep, class Period>
                bool try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);
            template<class Clock, class Duration>
                bool try_lock_shared_until(const chrono::time_point<Clock, Duration>& abs_time);
            void unlock_shared();
        }
    }
}
```

1 The class `shared_timed_mutex` provides a non-recursive mutex with shared ownership semantics.
2 The class `shared_timed_mutex` meets all of the shared timed mutex requirements (32.5.4.5). It is a standard-layout class (11.2).
3 The behavior of a program is undefined if:
   (3.1) it destroys a `shared_timed_mutex` object owned by any thread,
   (3.2) a thread attempts to recursively gain any ownership of a `shared_timed_mutex`, or
   (3.3) a thread terminates while possessing any ownership of a `shared_timed_mutex`.

32.5.5 Locks [thread.lock]

32.5.5.1 General [thread.lock.general]

1 A `lock` is an object that holds a reference to a lockable object and may unlock the lockable object during the lock's destruction (such as when leaving block scope). An execution agent may use a lock to aid in managing ownership of a lockable object in an exception safe manner. A lock is said to own a lockable object if it is
currently managing the ownership of that lockable object for an execution agent. A lock does not manage
the lifetime of the lockable object it references.

[Note 1: Locks are intended to ease the burden of unlocking the lockable object under both normal and exceptional
circumstances. — end note]

Some lock constructors take tag types which describe what should be done with the lockable object during
the lock’s construction.

```cpp
namespace std {
    struct defer_lock_t { }; // do not acquire ownership of the mutex
    struct try_to_lock_t { }; // try to acquire ownership of the mutex
        // without blocking
    struct adopt_lock_t { }; // assume the calling thread has already
        // obtained mutex ownership and manage it

    inline constexpr defer_lock_t defer_lock { };
    inline constexpr try_to_lock_t try_to_lock { };  
    inline constexpr adopt_lock_t adopt_lock { };  
}
```

### 32.5.5.2 Class template lock_guard

An object of type `lock_guard` controls the ownership of a lockable object within a scope. A `lock_guard` object
maintains ownership of a lockable object throughout the `lock_guard` object’s lifetime (6.7.3). The behavior
of a program is undefined if the lockable object referenced by `pm` does not exist for the entire lifetime of the
`lock_guard` object. The supplied `Mutex` type shall meet the `Cpp17BasicLockable` requirements (32.2.5.2).

```cpp
explicit lock_guard(mutex_type& m);
2
Preconditions: If `mutex_type` is not a recursive mutex, the calling thread does not own the mutex `m`.
3
Effects: Initializes `pm` with `m`. Calls `m.lock()`.

lock_guard(mutex_type& m, adopt_lock_t);
4
Preconditions: The calling thread owns the mutex `m`.
5
Effects: Initializes `pm` with `m`.
6
Throws: Nothing.

~lock_guard();
7
Effects: As if by `pm.unlock()`.
```

### 32.5.5.3 Class template scoped_lock

```cpp
namespace std {
    template<class... MutexTypes>
    class scoped_lock {
        public:
            using mutex_type = Mutex;  // If MutexTypes... consists of the single type Mutex
```
An object of type `scoped_lock` controls the ownership of lockable objects within a scope. A `scoped_lock` object maintains ownership of lockable objects throughout the `scoped_lock` object's lifetime (6.7.3). The behavior of a program is undefined if the lockable objects referenced by `pm` do not exist for the entire lifetime of the `scoped_lock` object. When `sizeof...(MutexTypes)` is 1, the supplied `Mutex` type shall meet the `Cpp17BasicLockable` requirements (32.2.5.2). Otherwise, each of the mutex types shall meet the `Cpp17Lockable` requirements (32.2.5.3).

```cpp
explicit scoped_lock(MutexTypes&... m);
```

**Preconditions:** If a `MutexTypes` type is not a recursive mutex, the calling thread does not own the corresponding mutex element of `m`.

**Effects:** Initializes `pm` with `tie(m...)`. Then if `sizeof...(MutexTypes)` is 0, no effects. Otherwise if `sizeof...(MutexTypes)` is 1, then `m.lock()`. Otherwise, `lock(m...)`.

```cpp
explicit scoped_lock(adopt_lock_t, MutexTypes&... m);
```

**Preconditions:** The calling thread owns all the mutexes in `m`.

**Effects:** Initializes `pm` with `tie(m...)`.

**Throws:** Nothing.

```cpp
~scoped_lock();
```

**Effects:** For all `i` in `[0, sizeof...(MutexTypes))`, `get<i>(pm).unlock()`.

### 32.5.5.4 Class template unique_lock

#### 32.5.5.4.1 General

```cpp
namespace std {
    template<class Mutex>
    class unique_lock {
        using mutex_type = Mutex;

        // 32.5.5.4.2. construct/copy/destroy
        unique_lock() noexcept;
        explicit unique_lock(mutex_type& m);
        unique_lock(mutex_type& m, defer_lock_t) noexcept;
        unique_lock(mutex_type& m, try_to_lock_t);
        unique_lock(mutex_type& m, adopt_lock_t);
        template<class Clock, class Duration>
        unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
        template<class Rep, class Period>
        unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
    }

    unique_lock(const unique_lock&) = delete;
    unique_lock& operator=(const unique_lock&) = delete;

    unique_lock(unique_lock&& u) noexcept;
    unique_lock& operator=(unique_lock&& u);

    // 32.5.5.4.3, locking
    void lock();
}
```
bool try_lock();

template<class Rep, class Period>
  bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);

template<class Clock, class Duration>
  bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

void unlock();

// 32.5.5.4.4, modifiers
void swap(unique_lock& u) noexcept;
mutex_type* release() noexcept;

// 32.5.5.4.5, observers
bool owns_lock() const noexcept;
explicit operator bool () const noexcept;
mutex_type* mutex() const noexcept;

private:
  mutex_type* pm; // exposition only
  bool owns; // exposition only
};

template<class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;

An object of type unique_lock controls the ownership of a lockable object within a scope. Ownership of the lockable object may be acquired at construction or after construction, and may be transferred, after acquisition, to another unique_lock object. Objects of type unique_lock are not copyable but are movable. The behavior of a program is undefined if the contained pointer pm is not null and the lockable object pointed to by pm does not exist for the entire remaining lifetime (6.7.3) of the unique_lock object. The supplied Mutex type shall meet the Cpp17BasicLockable requirements (32.2.5.2).

[Note 1: unique_lock<Mutex> meets the Cpp17BasicLockable requirements. If Mutex meets the Cpp17Lockable requirements (32.2.5.3), unique_lock<Mutex> also meets the Cpp17Lockable requirements; if Mutex meets the Cpp17TimedLockable requirements (32.2.5.4), unique_lock<Mutex> also meets the Cpp17TimedLockable requirements. —end note]

32.5.5.4.2 Constructors, destructor, and assignment

unique_lock() noexcept;

Postconditions: pm == 0 and owns == false.

explicit unique_lock(mutex_type& m);

Preconditions: If mutex_type is not a recursive mutex the calling thread does not own the mutex.

Effects: Calls m.lock().

Postconditions: pm == addressof(m) and owns == true.

unique_lock(mutex_type& m, defer_lock_t) noexcept;

Postconditions: pm == addressof(m) and owns == false.

unique_lock(mutex_type& m, try_to_lock_t);

Preconditions: The supplied Mutex type meets the Cpp17Lockable requirements (32.2.5.3). If mutex_-type is not a recursive mutex the calling thread does not own the mutex.

Effects: Calls m.try_lock().

Postconditions: pm == addressof(m) and owns == res, where res is the value returned by the call to m.try_lock().

unique_lock(mutex_type& m, adopt_lock_t);

Preconditions: The calling thread owns the mutex.
Postconditions: pm == addressof(m) and owns == true.

Throws: Nothing.

```
template<class Clock, class Duration>
unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
```

Preconditions: If mutex_type is not a recursive mutex the calling thread does not own the mutex. The supplied Mutex type meets the Cpp17TimedLockable requirements (32.2.5.4).

Effects: Calls m.try_lock_until(abs_time).

Postconditions: pm == addressof(m) and owns == res, where res is the value returned by the call to m.try_lock_until(abs_time).

```
template<class Rep, class Period>
unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
```

Preconditions: If mutex_type is not a recursive mutex the calling thread does not own the mutex. The supplied Mutex type meets the Cpp17TimedLockable requirements (32.2.5.4).

Effects: Calls m.try_lock_for(rel_time).

Postconditions: pm == addressof(m) and owns == res, where res is the value returned by the call to m.try_lock_for(rel_time).

unique_lock(unique_lock&& u) noexcept;

```
Postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this construction), u.pm == 0 and u.owns == false.
```

unique_lock& operator=(unique_lock&& u);

```
Effects: If owns calls pm->unlock().
Postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this construction), u.pm == 0 and u.owns == false.
```

[Note 1: With a recursive mutex it is possible for both *this and u to own the same mutex before the assignment. In this case, *this will own the mutex after the assignment and u will not. — end note]

```
~unique_lock();
Effects: If owns calls pm->unlock().
```

§ 32.5.5.4.3 Locking

```
void lock();
Effects: As if by pm->lock().
Postconditions: owns == true.
Throws: Any exception thrown by pm->lock(). system_error when an exception is required (32.2.2).
Error conditions:
(4.1) — operation_not_permitted — if pm is nullptr.
(4.2) — resource_deadlock_would_occur — if on entry owns is true.
```

```
bool try_lock();
Preconditions: The supplied Mutex meets the Cpp17Lockable requirements (32.2.5.3).
Effects: As if by pm->try_lock().
Postconditions: owns == res, where res is the value returned by the call to try_lock().
Returns: The value returned by the call to try_lock().
Throws: Any exception thrown by pm->try_lock(). system_error when an exception is required (32.2.2).
```

Error conditions:
template<class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

---

### Preconditions:
The supplied Mutex type meets the `Cpp17TimedLockable` requirements (32.2.5.4).

### Effects:
As if by `pm->try_lock_until(abs_time)`.

### Postconditions:
`owns == res`, where `res` is the value returned by the call to `try_lock_until(abs_time)`.

### Returns:
The value returned by the call to `try_lock_until(abs_time)`.

### Error conditions:
- `operation_not_permitted` — if `pm` is `nullptr`.
- `resource_deadlock_would_occur` — if on entry `owns` is true.

---

template<class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);

---

### Preconditions:
The supplied Mutex type meets the `Cpp17TimedLockable` requirements (32.2.5.4).

### Effects:
As if by `pm->try_lock_for(rel_time)`.

### Postconditions:
`owns == res`, where `res` is the value returned by the call to `try_lock_for(rel_time)`.

### Returns:
The value returned by the call to `try_lock_for(rel_time)`.

### Error conditions:
- `operation_not_permitted` — if `pm` is `nullptr`.
- `resource_deadlock_would_occur` — if on entry `owns` is true.

---

void unlock();

---

### Effects:
As if by `pm->unlock()`.

### Postconditions:
`owns == false`.

### Throws:
`system_error` when an exception is required (32.2.2).

### Error conditions:
- `operation_not_permitted` — if on entry `owns` is false.
- `resource_deadlock_would_occur` — if on entry `owns` is true.

---

### Modifiers

`void swap(unique_lock& u) noexcept;`

### Effects:
Swaps the data members of `*this` and `u`.

`mutex_type* release() noexcept;`

### Postconditions:
`pm == 0` and `owns == false`.

### Returns:
The previous value of `pm`.

---

`template<class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;`

### Effects:
As if by `x.swap(y)`.

---

### Observers

`bool owns_lock() const noexcept;`

### Returns:
`owns`.
explicit operator bool() const noexcept;

Returns: owns.

mutex_type *mutex() const noexcept;

Returns: pm.

32.5.5.5 Class template shared_lock

32.5.5.5.1 General

namespace std {

template<class Mutex>

class shared_lock {

public:
    using mutex_type = Mutex;

    // 32.5.5.5.2, construct/copy/destroy
    shared_lock() noexcept;
    explicit shared_lock(mutex_type& m); // blocking
    shared_lock(mutex_type& m, defer_lock_t) noexcept;
    shared_lock(mutex_type& m, try_to_lock_t);
    shared_lock(mutex_type& m, adopt_lock_t);
    template<class Clock, class Duration>
        shared_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
    template<class Rep, class Period>
        shared_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);

    ~shared_lock();

    shared_lock(const shared_lock&) = delete;
    shared_lock& operator=(const shared_lock&) = delete;

    shared_lock(shared_lock&& u) noexcept;
    shared_lock& operator=(shared_lock&& u) noexcept;

    // 32.5.5.5.3, locking
    void lock(); // blocking
    bool try_lock();
    template<class Rep, class Period>
        bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template<class Clock, class Duration>
        bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    // 32.5.5.5.4, modifiers
    void swap(shared_lock& u) noexcept;
    mutex_type* release() noexcept;

    // 32.5.5.5.5, observers
    bool owns_lock() const noexcept;
    explicit operator bool () const noexcept;
    mutex_type* mutex() const noexcept;

private:
    mutex_type* pm; // exposition only
    bool owns; // exposition only
};

    template<class Mutex>
        void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;

1 An object of type shared_lock controls the shared ownership of a lockable object within a scope. Shared
ownership of the lockable object may be acquired at construction or after construction, and may be transferred,
after acquisition, to another shared_lock object. Objects of type shared_lock are not copyable but are
movable. The behavior of a program is undefined if the contained pointer pm is not null and the lockable

§ 32.5.5.5.1
object pointed to by `pm` does not exist for the entire remaining lifetime (6.7.3) of the `shared_lock` object. The supplied `Mutex` type shall meet the shared mutex requirements (32.5.4.5).

[Note 1: `shared_lock<Mutex>` meets the Cpp17TimedLockable requirements (32.2.5.4). — end note]

### 32.5.5.5.2 Constructors, destructor, and assignment

```cpp
shared_lock() noexcept;
```

**Postconditions:** `pm == nullptr` and `owns == false`.

```cpp
explicit shared_lock(mutex_type& m);
```

**Preconditions:** The calling thread does not own the mutex for any ownership mode.

**Effects:** Calls `m.lock_shared()`.

**Postconditions:** `pm == addressof(m)` and `owns == true`.

```cpp
shared_lock(mutex_type& m, defer_lock_t) noexcept;
```

**Postconditions:** `pm == addressof(m)` and `owns == false`.

```cpp
shared_lock(mutex_type& m, try_to_lock_t);
```

**Preconditions:** The calling thread does not own the mutex for any ownership mode.

**Effects:** Calls `m.try_lock_shared()`.

**Postconditions:** `pm == addressof(m)` and `owns == res` where `res` is the value returned by the call to `m.try_lock_shared()`.

```cpp
shared_lock(mutex_type& m, adopt_lock_t);
```

**Preconditions:** The calling thread has shared ownership of the mutex.

**Postconditions:** `pm == addressof(m)` and `owns == true`.

```cpp
template<class Clock, class Duration>
shared_lock(mutex_type& m,
            const chrono::time_point<Clock, Duration>& abs_time);
```

**Preconditions:** The calling thread does not own the mutex for any ownership mode.

**Effects:** Calls `m.try_lock_shared_until(abs_time)`.

**Postconditions:** `pm == addressof(m)` and `owns == res` where `res` is the value returned by the call to `m.try_lock_shared_until(abs_time)`.

```cpp
template<class Rep, class Period>
shared_lock(mutex_type& m,
            const chrono::duration<Rep, Period>& rel_time);
```

**Preconditions:** The calling thread does not own the mutex for any ownership mode.

**Effects:** Calls `m.try_lock_shared_for(rel_time)`.

**Postconditions:** `pm == addressof(m)` and `owns == res` where `res` is the value returned by the call to `m.try_lock_shared_for(rel_time)`.

~`shared_lock();`

**Effects:** If `owns` calls `pm->unlock_shared()`.

```cpp
shared_lock(shared_lock&& sl) noexcept;
```

**Postconditions:** `pm == sl_p.pm` and `owns == sl_p.owns` (where `sl_p` is the state of `sl` just prior to this construction), `sl_p.pm == nullptr` and `sl.owns == false`.

```cpp
shared_lock& operator=(shared_lock&& sl) noexcept;
```

**Effects:** If `owns` calls `pm->unlock_shared()`.

**Postconditions:** `pm == sl_p.pm` and `owns == sl_p.owns` (where `sl_p` is the state of `sl` just prior to this assignment), `sl_p.pm == nullptr` and `sl.owns == false`. 
32.5.5.5.3 Locking

void lock();

   Effects: As if by pm->lock_shared().
   Postconditions: owns == true.
   Throws: Any exception thrown by pm->lock_shared(). `system_error` when an exception is required (32.2.2).

   Error conditions:
   - operation_not_permitted — if pm is `nullptr`.
   - resource_deadlock_would_occur — if on entry owns is true.

bool try_lock();

   Effects: As if by pm->try_lock_shared().
   Postconditions: owns == res, where res is the value returned by the call to pm->try_lock_shared().
   Returns: The value returned by the call to pm->try_lock_shared().
   Throws: Any exception thrown by pm->try_lock_shared(). `system_error` when an exception is required (32.2.2).

   Error conditions:
   - operation_not_permitted — if pm is `nullptr`.
   - resource_deadlock_would_occur — if on entry owns is true.

template<class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

   Effects: As if by pm->try_lock_shared_until(abs_time).
   Postconditions: owns == res, where res is the value returned by the call to pm->try_lock_shared_until(abs_time).
   Returns: The value returned by the call to pm->try_lock_shared_until(abs_time).
   Throws: Any exception thrown by pm->try_lock_shared_until(abs_time). `system_error` when an exception is required (32.2.2).

   Error conditions:
   - operation_not_permitted — if pm is `nullptr`.
   - resource_deadlock_would_occur — if on entry owns is true.

template<class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);

   Effects: As if by pm->try_lock_shared_for(rel_time).
   Postconditions: owns == res, where res is the value returned by the call to pm->try_lock_shared_for(rel_time).
   Returns: The value returned by the call to pm->try_lock_shared_for(rel_time).
   Throws: Any exception thrown by pm->try_lock_shared_for(rel_time). `system_error` when an exception is required (32.2.2).

   Error conditions:
   - operation_not_permitted — if pm is `nullptr`.
   - resource_deadlock_would_occur — if on entry owns is true.

void unlock();

   Effects: As if by pm->unlock_shared().
   Postconditions: owns == false.
   Throws: `system_error` when an exception is required (32.2.2).
Error conditions:
— operation_not_permitted — if on entry owns is false.

### 32.5.5.4 Modifiers

```cpp
void swap(shared_lock&amp; sl) noexcept;
```

**Effects**: Swaps the data members of *this and sl.

```cpp
mutex_type* release() noexcept;
```

**Postconditions**: pm == nullptr and owns == false.

**Returns**: The previous value of pm.

```cpp
template<class Mutex>
void swap(shared_lock&lt;Mutex&gt;& x, shared_lock&lt;Mutex&gt;& y) noexcept;
```

**Effects**: As if by `x.swap(y)`.

### 32.5.5.5 Observers

```cpp
bool owns_lock() const noexcept;
```

**Returns**: owns.

```cpp
explicit operator bool() const noexcept;
```

**Returns**: owns.

```cpp
mutex_type* mutex() const noexcept;
```

**Returns**: pm.

### 32.5.6 Generic locking algorithms

```cpp
template<class L1, class L2, class... L3> int try_lock(L1&amp;, L2&amp;, L3&amp;...);
```

**Preconditions**: Each template parameter type meets the `Cpp17Lockable` requirements.

**Note 1**: The `unique_lock` class template meets these requirements when suitably instantiated. — end note

**Effects**: Calls `try_lock()` for each argument in order beginning with the first until all arguments have been processed or a call to `try_lock()` fails, either by returning `false` or by throwing an exception. If a call to `try_lock()` fails, `unlock()` is called for all prior arguments with no further calls to `try_lock()`.

**Returns**: -1 if all calls to `try_lock()` returned `true`, otherwise a zero-based index value that indicates the argument for which `try_lock()` returned `false`.

```cpp
template<class L1, class L2, class... L3> void lock(L1&amp;, L2&amp;, L3&amp;...);
```

**Preconditions**: Each template parameter type meets the `Cpp17Lockable` requirements.

**Note 2**: The `unique_lock` class template meets these requirements when suitably instantiated. — end note

**Effects**: All arguments are locked via a sequence of calls to `lock()`, `try_lock()`, or `unlock()` on each argument. The sequence of calls does not result in deadlock, but is otherwise unspecified.

**Note 3**: A deadlock avoidance algorithm such as try-and-back-off can be used, but the specific algorithm is not specified to avoid over-constraining implementations. — end note

If a call to `lock()` or `try_lock()` throws an exception, `unlock()` is called for any argument that had been locked by a call to `lock()` or `try_lock()`.

### 32.5.7 Call once

```cpp
namespace std {
    struct once_flag {
        constexpr once_flag() noexcept;
    }
}
```

**32.5.7.1 Struct once_flag**

```cpp
namespace std {
    struct once_flag {
        constexpr once_flag() noexcept;
    }
}
```
The class `once_flag` is an opaque data structure that `call_once` uses to initialize data without causing a data race or deadlock.

```cpp
constexpr once_flag() noexcept;
```

**Synchronization:** The construction of a `once_flag` object is not synchronized.

**Postconditions:** The object's internal state is set to indicate to an invocation of `call_once` with the object as its initial argument that no function has been called.

32.5.7.2 Function `call_once` [thread.once.callonce]

```cpp
template<class Callable, class... Args>
void call_once(once_flag& flag, Callable&& func, Args&&... args);
```

**Mandates:** `is_invocable_v<Callable, Args...>` is true.

**Effects:** An execution of `call_once` that does not call its `func` is a passive execution. An execution of `call_once` that calls its `func` is an active execution. An active execution calls `INVOKE(std::forward<Callable>(func), std::forward<Args>(args)...).` If such a call to `func` throws an exception the execution is exceptional, otherwise it is returning. An exceptional execution propagates the exception to the caller of `call_once`. Among all executions of `call_once` for any given `once_flag`: at most one is a returning execution; if there is a returning execution, it is the last active execution; and there are passive executions only if there is a returning execution.

[Note 1: Passive executions allow other threads to reliably observe the results produced by the earlier returning execution. — end note]

**Synchronization:** For any given `once_flag`: all active executions occur in a total order; completion of an active execution synchronizes with (6.9.2) the start of the next one in this total order; and the returning execution synchronizes with the return from all passive executions.

**Throws:** `system_error` when an exception is required (32.2.2), or any exception thrown by `func`.

**Example 1:**

```cpp
// global flag, regular function
void init();
std::once_flag flag;

void f() {
  std::call_once(flag, init);
}

// function static flag, function object
struct initializer {
  void operator()();
};

void g() {
  static std::once_flag flag2;
  std::call_once(flag2, initializer());
}

// object flag, member function
class information {
  std::once_flag verified;
  void verifier();
  public:
    void verify() { std::call_once(verified, &information::verifier, *this); }
};
```

— end example

§ 32.5.7.2
32.6 Condition variables

32.6.1 General

1 Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached. Class condition_variable provides a condition variable that can only wait on an object of type unique_lock<mutex>, allowing the implementation to be more efficient. Class condition_variable_any provides a general condition variable that can wait on objects of user-supplied lock types.

2 Condition variables permit concurrent invocation of the wait, wait_for, wait_until, notify_one and notify_all member functions.

3 The executions of notify_one and notify_all are atomic. The executions of wait, wait_for, and wait_until are performed in three atomic parts:
   1. the release of the mutex and entry into the waiting state;
   2. the unblocking of the wait; and
   3. the reacquisition of the lock.

4 The implementation behaves as if all executions of notify_one, notify_all, and each part of the wait, wait_for, and wait_until executions are executed in a single unspecified total order consistent with the 'happens before' order.

5 Condition variable construction and destruction need not be synchronized.

32.6.2 Header <condition_variable> synopsis

namespace std {
    class condition_variable;
    class condition_variable_any;

    void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);

    enum class cv_status { no_timeout, timeout };
}

32.6.3 Non-member functions

void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);

1 Preconditions: lk is locked by the calling thread and either
   (1.1) — no other thread is waiting on cond, or
   (1.2) — lk.mutex() returns the same value for each of the lock arguments supplied by all concurrently
          waiting (via wait, wait_for, or wait_until) threads.

2 Effects: Transfers ownership of the lock associated with lk into internal storage and schedules cond to
be notified when the current thread exits, after all objects of thread storage duration associated with
the current thread have been destroyed. This notification is equivalent to:

lk.unlock();
cond.notify_all();

3 Synchronization: The implied lk.unlock() call is sequenced after the destruction of all objects with
thread storage duration associated with the current thread.

4 [Note 1: The supplied lock is held until the thread exits, which might cause deadlock due to lock ordering
issues. — end note]

5 [Note 2: It is the user’s responsibility to ensure that waiting threads do not erroneously assume that the thread
has finished if they experience spurious wakeups. This typically requires that the condition being waited for
is satisfied while holding the lock on lk, and that this lock is not released and reacquired prior to calling
notify_all_at_thread_exit. — end note]

32.6.4 Class condition_variable

namespace std {
    class condition_variable {
        public:
            condition_variable();
    }

The class `condition_variable` is a standard-layout class (11.2).

```cpp
void notify_one() noexcept;
void notify_all() noexcept;
void wait(unique_lock<mutex>& lock);
template<class Predicate>
  void wait(unique_lock<mutex>& lock, Predicate pred);
template<class Clock, class Duration>
  cv_status wait_until(unique_lock<mutex>& lock,
                       const chrono::time_point<Clock, Duration>& abs_time);
template<class Clock, class Duration, class Predicate>
  bool wait_until(unique_lock<mutex>& lock,
                  const chrono::time_point<Clock, Duration>& abs_time,
                  Predicate pred);
template<class Rep, class Period>
  cv_status wait_for(unique_lock<mutex>& lock,
                     const chrono::duration<Rep, Period>& rel_time);
template<class Rep, class Period, class Predicate>
  bool wait_for(unique_lock<mutex>& lock,
                const chrono::duration<Rep, Period>& rel_time,
                Predicate pred);
```

Using `native_handle_type = implementation-defined;` // see 32.2.3

```cpp
native_handle_type native_handle(); // see 32.2.3
};
```

1. The class `condition_variable` is a standard-layout class (11.2).

```cpp
condition_variable();
```

2. Throwing: `system_error` when an exception is required (32.2.2).

3. Error conditions:

   - `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.

```cpp
~condition_variable();
```

4. Preconditions: There is no thread blocked on `*this`.

   [Note 1: That is, all threads have been notified; they could subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction. Undefined behavior ensues if a thread waits on `*this` once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of `wait, wait_for,` or `wait_until` that take a predicate. — end note]

```cpp
void notify_one() noexcept;
```

5. Effects: If any threads are blocked waiting for `*this`, unblocks one of those threads.

```cpp
void notify_all() noexcept;
```

6. Effects: Unblocks all threads that are blocked waiting for `*this`.

```cpp
void wait(unique_lock<mutex>& lock);
```

7. Preconditions: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

   - no other thread is waiting on this `condition_variable` object or
   - `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait, wait_for,` or `wait_until`) threads.

8. Effects:

   - Atomically calls `lock.unlock()` and blocks on `*this`. 

§ 32.6.4
— When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.

— The function will unblock when signaled by a call to notify_one() or a call to notify_all(), or spuriously.

Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

Throws: Nothing.

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.2).

[Note 2: This can happen if the re-locking of the mutex throws an exception. — end note]

```cpp
template<class Predicate>
void wait(unique_lock<mutex>& lock, Predicate pred);
```

Preconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

— no other thread is waiting on this condition_variable object or

— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.

Effects: Equivalent to:

```cpp
while (!pred())
    wait(lock);
```

Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

Throws: Any exception thrown by pred.

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.2).

[Note 3: This can happen if the re-locking of the mutex throws an exception. — end note]

```cpp
template<class Clock, class Duration>
cv_status wait_until(unique_lock<mutex>& lock,
                     const chrono::time_point<Clock, Duration>& abs_time);
```

Preconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

— no other thread is waiting on this condition_variable object or

— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.

Effects:

— Atomically calls lock.unlock() and blocks on *this.

— When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.

— The function will unblock when signaled by a call to notify_one(), a call to notify_all(), expiration of the absolute timeout (32.2.4) specified by abs_time, or spuriously.

— If the function exits via an exception, lock.lock() is called prior to exiting the function.

Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

Returns: cv_status::timeout if the absolute timeout (32.2.4) specified by abs_time expired, otherwise cv_status::no_timeout.

Throws: Timeout-related exceptions (32.2.4).

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.2).

[Note 4: This can happen if the re-locking of the mutex throws an exception. — end note]

```cpp
template<class Rep, class Period>
cv_status wait_for(unique_lock<mutex>& lock,
                    const chrono::duration<Rep, Period>& rel_time);
```

Preconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

— no other thread is waiting on this condition_variable object or

— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.
Effects: Equivalent to:

```cpp
return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

Postconditions: `lock.owns_lock()` is true and `lock.mutex()` is locked by the calling thread.

Returns: `cv_status::timeout` if the relative timeout (32.2.4) specified by `rel_time` expired, otherwise `cv_status::no_timeout`.

Throws: Timeout-related exceptions (32.2.4).

Remarks: If the function fails to meet the postcondition, `terminate()` is called (14.6.2).

[Note 5: This can happen if the re-locking of the mutex throws an exception. — end note]

```cpp
template<class Clock, class Duration, class Predicate>
bool wait_until(unique_lock<mutex>& lock,
    const chrono::time_point<Clock, Duration>& abs_time,
    Predicate pred);
```

Preconditions: `lock.owns_lock()` is true and `lock.mutex()` is locked by the calling thread, and either

- no other thread is waiting on this `condition_variable` object or
- `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

Effects: Equivalent to:

```cpp
while (!pred())
    if (wait_until(lock, abs_time) == cv_status::timeout)
        return pred();
return true;
```

Postconditions: `lock.owns_lock()` is true and `lock.mutex()` is locked by the calling thread.

[Note 6: The returned value indicates whether the predicate evaluated to `true` regardless of whether the timeout was triggered. — end note]

Throws: Timeout-related exceptions (32.2.4) or any exception thrown by `pred`.

Remarks: If the function fails to meet the postcondition, `terminate()` is called (14.6.2).

[Note 7: This can happen if the re-locking of the mutex throws an exception. — end note]

```cpp
template<class Rep, class Period, class Predicate>
bool wait_for(unique_lock<mutex>& lock,
    const chrono::duration<Rep, Period>& rel_time,
    Predicate pred);
```

Preconditions: `lock.owns_lock()` is true and `lock.mutex()` is locked by the calling thread, and either

- no other thread is waiting on this `condition_variable` object or
- `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

Effects: Equivalent to:

```cpp
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
```

Postconditions: `lock.owns_lock()` is true and `lock.mutex()` is locked by the calling thread.

[Note 8: There is no blocking if `pred()` is initially `true`, even if the timeout has already expired. — end note]

[Note 9: The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — end note]

Throws: Timeout-related exceptions (32.2.4) or any exception thrown by `pred`.

Remarks: If the function fails to meet the postcondition, `terminate()` is called (14.6.2).

[Note 10: This can happen if the re-locking of the mutex throws an exception. — end note]
[Note 1: All of the standard mutex types meet this requirement. If a Lock type other than one of the standard mutex types or a unique_lock wrapper for a standard mutex type is used with condition_variable_any, any necessary synchronization is assumed to be in place with respect to the predicate associated with the condition_variable_any instance. — end note]

namespace std {
    class condition_variable_any {
        public:
            condition_variable_any();
            ~condition_variable_any();

            condition_variable_any(const condition_variable_any&) = delete;
            condition_variable_any& operator=(const condition_variable_any&) = delete;

            void notify_one() noexcept;
            void notify_all() noexcept;

            // 32.6.5.2, noninterruptible waits
            template<class Lock>
                void wait(Lock& lock);
            template<class Lock, class Predicate>
                void wait(Lock& lock, Predicate pred);

            template<class Lock, class Clock, class Duration>
                cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
            template<class Lock, class Clock, class Duration, class Predicate>
                bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);

            template<class Lock, class Rep, class Period>
                cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
            template<class Lock, class Rep, class Period, class Predicate>
                bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);

            // 32.6.5.3, interruptible waits
            template<class Lock, class Predicate>
                bool wait(Lock& lock, stop_token stoken, Predicate pred);
            template<class Lock, class Clock, class Duration, class Predicate>
                bool wait_until(Lock& lock, stop_token stoken, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
            template<class Lock, class Rep, class Period, class Predicate>
                bool wait_for(Lock& lock, stop_token stoken, const chrono::duration<Rep, Period>& rel_time, Predicate pred);

        condition_variable_any();
    }
};

condition_variable_any();

2  Throws: bad_alloc or system_error when an exception is required (32.2.2).

3  Error conditions:

(3.1)  — resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

(3.2)  — operation_not_permitted — if the thread does not have the privilege to perform the operation.

~condition_variable_any();

4  Preconditions: There is no thread blocked on *this.

[Note 2: That is, all threads have been notified; they could subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction. Undefined behavior ensues if a thread waits on *this once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of wait, wait_for, or wait_until that take a predicate. — end note]

void notify_one() noexcept;

5  Effects: If any threads are blocked waiting for *this, unblocks one of those threads.
void notify_all() noexcept;

Effects: Unblocks all threads that are blocked waiting for *this.

32.6.5.2 Noninterruptible waits

```cpp
template<class Lock>
void wait(Lock& lock);
```

Effects:

1. Atomically calls lock.unlock() and blocks on *this.
2. When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.
3. The function will unblock when signaled by a call to notify_one(), a call to notify_all(), or spuriously.

Postconditions: lock is locked by the calling thread.

Throws: Nothing.

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.2).

[Note 1: This can happen if the re-locking of the mutex throws an exception. — end note]

```cpp
template<class Lock, class Predicate>
void wait(Lock& lock, Predicate pred);
```

Effects: Equivalent to:

```cpp
while (!pred())
    wait(lock);
```

```cpp
template<class Lock, class Clock, class Duration>
cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
```

Effects:

1. Atomically calls lock.unlock() and blocks on *this.
2. When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.
3. The function will unblock when signaled by a call to notify_one(), a call to notify_all(), expiration of the absolute timeout (32.2.4) specified by abs_time, or spuriously.
4. If the function exits via an exception, lock.lock() is called prior to exiting the function.

Postconditions: lock is locked by the calling thread.

Returns: cv_status::timeout if the absolute timeout (32.2.4) specified by abs_time expired, otherwise cv_status::no_timeout.

Throws: Timeout-related exceptions (32.2.4).

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.2).

[Note 2: This can happen if the re-locking of the mutex throws an exception. — end note]

```cpp
template<class Lock, class Rep, class Period>
cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
```

Effects: Equivalent to:

```cpp
return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

Postconditions: lock is locked by the calling thread.

Returns: cv_status::timeout if the relative timeout (32.2.4) specified by rel_time expired, otherwise cv_status::no_timeout.

Throws: Timeout-related exceptions (32.2.4).

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.2).

[Note 3: This can happen if the re-locking of the mutex throws an exception. — end note]
template<class Lock, class Clock, class Duration, class Predicate>
bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);

16 Effects: Equivalent to:
while (!pred())
  if (wait_until(lock, abs_time) == cv_status::timeout)
    return pred();
  return true;

[Note 4: There is no blocking if pred() is initially true, or if the timeout has already expired. — end note]
17 [Note 5: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered. — end note]

template<class Lock, class Rep, class Period, class Predicate>
bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);
19 Effects: Equivalent to:
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));

32.6.5.3 Interruptible waits  [thread.condvarany.intwait]
1 The following wait functions will be notified when there is a stop request on the passed stop_token. In that case the functions return immediately, returning false if the predicate evaluates to false.

template<class Lock, class Predicate>
bool wait(Lock& lock, stop_token stoken, Predicate pred);
2 Effects: Registers for the duration of this call *this to get notified on a stop request on stoken during this call and then equivalent to:
while (!stoken.stop_requested()) {
  if (pred())
    return true;
  wait(lock);
}
return pred();
3 [Note 1: The returned value indicates whether the predicate evaluated to true regardless of whether there was a stop request. — end note]
4 Postconditions: lock is locked by the calling thread.
5 Throws: Any exception thrown by pred.
6 Remarks: If the function fails to meet the postcondition, terminate is called (14.6.2).
[Note 2: This can happen if the re-locking of the mutex throws an exception. — end note]

template<class Lock, class Clock, class Duration, class Predicate>
bool wait_until(Lock& lock, stop_token stoken,
const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
7 Effects: Registers for the duration of this call *this to get notified on a stop request on stoken during this call and then equivalent to:
while (!stoken.stop_requested()) {
  if (pred())
    return true;
  if (wait_until(lock, abs_time) == cv_status::timeout)
    return pred();
}
return pred();
8 [Note 3: There is no blocking if pred() is initially true, stoken.stop_requested() was already true or the timeout has already expired. — end note]
[Note 4: The returned value indicates whether the predicate evaluated to true regardless of whether the timeout was triggered or a stop request was made. — end note]
9 Postconditions: lock is locked by the calling thread.
10 Throws: Timeout-related exceptions (32.2.4), or any exception thrown by pred.
Remarks: If the function fails to meet the postcondition, `terminate` is called (14.6.2).

[Note 5: This can happen if the re-locking of the mutex throws an exception. — end note]

```cpp
template<class Lock, class Rep, class Period, class Predicate>
bool wait_for(Lock& lock, stop_token stoken,
const chrono::duration<Rep, Period>& rel_time, Predicate pred);
```

Effects: Equivalent to:

```cpp
return wait_until(lock, std::move(stoken), chrono::steady_clock::now() + rel_time,
std::move(pred));
```

32.7 Semaphore

32.7.1 General

Semaphores are lightweight synchronization primitives used to constrain concurrent access to a shared resource. They are widely used to implement other synchronization primitives and, whenever both are applicable, can be more efficient than condition variables.

A counting semaphore is a semaphore object that models a non-negative resource count. A binary semaphore is a semaphore object that has only two states. A binary semaphore should be more efficient than the default implementation of a counting semaphore with a unit resource count.

32.7.2 Header `<semaphore>` synopsis

```cpp
namespace std {
    template<ptrdiff_t least_max_value = implementation-defined>
    class counting_semaphore;

    using binary_semaphore = counting_semaphore<1>;
}
```

32.7.3 Class template `counting_semaphore`

```cpp
namespace std {
    template<ptrdiff_t least_max_value = implementation-defined>
    class counting_semaphore {
        public:
            static constexpr ptrdiff_t max() noexcept;
            constexpr explicit counting_semaphore(ptrdiff_t desired);
            ~counting_semaphore();
            counting_semaphore(const counting_semaphore&) = delete;
            counting_semaphore& operator=(const counting_semaphore&) = delete;
            void release(ptrdiff_t update = 1);
            void acquire();
            bool try_acquire() noexcept;
        template<class Rep, class Period>
            bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
        template<class Clock, class Duration>
            bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);

        private:
            ptrdiff_t counter; // exposition only
    };
}
```

1 Class template `counting_semaphore` maintains an internal counter that is initialized when the semaphore is created. The counter is decremented when a thread acquires the semaphore, and is incremented when a thread releases the semaphore. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

2 `least_max_value` shall be non-negative; otherwise the program is ill-formed.

3 Concurrent invocations of the member functions of `counting semaphore`, other than its destructor, do not introduce data races.

§ 32.7.3
static constexpr ptrdiff_t max() noexcept;

Returns: The maximum value of counter. This value is greater than or equal to \texttt{least\_max\_value}.

constexpr explicit counting_semaphore(ptrdiff_t desired);

Preconditions: \texttt{desired} \(\geq\) 0 is true, and \texttt{desired} \(\leq\) \texttt{max()} is true.

Effects: Initializes counter with desired.

Throws: Nothing.

void release(ptrdiff_t update = 1);

Preconditions: \texttt{update} \(\geq\) 0 is true, and \texttt{update} \(\leq\) \texttt{max()} - counter is true.

Effects: Atomically execute \texttt{counter += update}. Then, unblocks any threads that are waiting for counter to be greater than zero.

Synchronization: Strongly happens before invocations of \texttt{try\_acquire} that observe the result of the effects.

Throws: \texttt{system\_error} when an exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.4.2).

bool try_acquire() noexcept;

Effects: Attempts to atomically decrement counter if it is positive, without blocking. If counter is not decremented, there is no effect and \texttt{try\_acquire} immediately returns. An implementation may fail to decrement counter even if it is positive.

[Note 1: This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare and exchange (Clause 31). — end note]

An implementation should ensure that \texttt{try\_acquire} does not consistently return \texttt{false} in the absence of contending semaphore operations.

Returns: \texttt{true} if counter was decremented, otherwise \texttt{false}.

void acquire();

Effects: Repeatedly performs the following steps, in order:

— Evaluates \texttt{try\_acquire}. If the result is \texttt{true}, returns.

— Blocks on *this until counter is greater than zero.

Throws: \texttt{system\_error} when an exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.4.2).

```cpp
template<class Rep, class Period>
bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);

template<class Clock, class Duration>
bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Effects: Repeatedly performs the following steps, in order:

— Evaluates \texttt{try\_acquire()}. If the result is \texttt{true}, returns \texttt{true}.

— Blocks on *this until counter is greater than zero or until the timeout expires. If it is unblocked by the timeout expiring, returns \texttt{false}.

The timeout expires (32.2.4) when the current time is after \texttt{abs\_time} (for \texttt{try\_acquire\_until}) or when at least \texttt{rel\_time} has passed from the start of the function (for \texttt{try\_acquire\_for}).

Throws: Timeout-related exceptions (32.2.4), or \texttt{system\_error} when a non-timeout-related exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.4.2).

32.8 Coordination types [thread.coord]

32.8.1 General [thread.coord.general]

Subclause 32.8 describes various concepts related to thread coordination, and defines the coordination types \texttt{latch} and \texttt{barrier}. These types facilitate concurrent computation performed by a number of threads.
32.8.2 Latches

32.8.2.1 General

1 A latch is a thread coordination mechanism that allows any number of threads to block until an expected number of threads arrive at the latch (via the `count_down` function). The expected count is set when the latch is created. An individual latch is a single-use object; once the expected count has been reached, the latch cannot be reused.

32.8.2.2 Header `<latch>` synopsis

```cpp
namespace std {
    class latch;
}
```

32.8.2.3 Class latch

```cpp
namespace std {
    class latch {
        public:
            static constexpr ptrdiff_t max() noexcept;
            constexpr explicit latch(ptrdiff_t expected);
            ~latch();
            latch(const latch&) = delete;
            latch& operator=(const latch&) = delete;
            void count_down(ptrdiff_t update = 1);
            bool try_wait() const noexcept;
            void wait() const;
            void arrive_and_wait(ptrdiff_t update = 1);

        private:
            ptrdiff_t counter; // exposition only
    };
}
```

1 A `latch` maintains an internal counter that is initialized when the latch is created. Threads can block on the latch object, waiting for counter to be decremented to zero.

2 Concurrent invocations of the member functions of `latch`, other than its destructor, do not introduce data races.

    static constexpr ptrdiff_t max() noexcept;

3 Returns: The maximum value of `counter` that the implementation supports.

    constexpr explicit latch(ptrdiff_t expected);

4 Preconditions: `expected >= 0` is true and `expected <= max()` is true.

5 Effects: Initializes `counter` with `expected`.

6 Throws: Nothing.

    void count_down(ptrdiff_t update = 1);

7 Preconditions: `update >= 0` is true, and `update <= counter` is true.

8 Effects: Atomically decrements `counter` by `update`. If `counter` is equal to zero, unblocks all threads blocked on `*this`.

9 Synchronization: Strongly happens before the returns from all calls that are unblocked.

10 Throws: `system_error` when an exception is required (32.2.2).

11 Error conditions: Any of the error conditions allowed for mutex types (32.5.4.2).

    bool try_wait() const noexcept;

12 Returns: With very low probability `false`. Otherwise `counter == 0`.  

§ 32.8.2.3
void wait() const;
13
Effects: If counter equals zero, returns immediately. Otherwise, blocks on *this until a call to
count_down that decrements counter to zero.
14
Throws: system_error when an exception is required (32.2.2).
15
Error conditions: Any of the error conditions allowed for mutex types (32.5.4.2).

void arrive_and_wait(ptrdiff_t update = 1);
16
Effects: Equivalent to:
    count_down(update);
    wait();

32.8.3 Barriers

32.8.3.1 General

A barrier is a thread coordination mechanism whose lifetime consists of a sequence of barrier phases, where
each phase allows at most an expected number of threads to block until the expected number of threads
arrive at the barrier.

[Note 1: A barrier is useful for managing repeated tasks that are handled by multiple threads. — end note]

32.8.3.2 Header <barrier> synopsis

namespace std {
    template<class CompletionFunction = see below>
    class barrier;
}

32.8.3.3 Class template barrier

namespace std {
    template<class CompletionFunction = see below>
    class barrier {
    public:
        using arrival_token = see below;

        static constexpr ptrdiff_t max() noexcept;
        constexpr explicit barrier(ptrdiff_t expected,
            CompletionFunction f = CompletionFunction());
        ~barrier();
        barrier(const barrier&) = delete;
        barrier& operator=(const barrier&) = delete;

        [[nondiscard]] arrival_token arrive(ptrdiff_t update = 1);
        void wait(arrival_token& arrival) const;
        void arrive_and_wait();
        void arrive_and_drop();

    private:
        CompletionFunction completion;         // exposition only
    };
}

1 Each barrier phase consists of the following steps:

(1.1) — The expected count is decremented by each call to arrive or arrive_and_drop.

(1.2) — When the expected count reaches zero, the phase completion step is run. For the specialization with
the default value of the CompletionFunction template parameter, the completion step is run as part
of the call to arrive or arrive_and_drop that caused the expected count to reach zero. For other
specializations, the completion step is run on one of the threads that arrived at the barrier during the phase.
When the completion step finishes, the expected count is reset to what was specified by the `expected` argument to the constructor, possibly adjusted by calls to `arrive_and_drop`, and the next phase starts.

Each phase defines a **phase synchronization point**. Threads that arrive at the barrier during the phase can block on the phase synchronization point by calling `wait`, and will remain blocked until the phase completion step is run.

The **phase completion step** that is executed at the end of each phase has the following effects:

3.1 Invokes the completion function, equivalent to `completion()`.

3.2 Unblocks all threads that are blocked on the phase synchronization point.

The end of the completion step strongly happens before the returns from all calls that were unblocked by the completion step. For specializations that do not have the default value of the `CompletionFunction` template parameter, the behavior is undefined if any of the barrier object’s member functions other than `wait` are called while the completion step is in progress.

Concurrent invocations of the member functions of `barrier`, other than its destructor, do not introduce data races. The member functions `arrive` and `arrive_and_drop` execute atomically.

`CompletionFunction` shall meet the `Cpp17MoveConstructible` (Table 28) and `Cpp17Destructible` (Table 32) requirements. `is_nothrow_invocable_v<CompletionFunction&>` shall be `true`.

The default value of the `CompletionFunction` template parameter is an unspecified type, such that, in addition to satisfying the requirements of `CompletionFunction`, it meets the `Cpp17DefaultConstructible` requirements (Table 27) and `completion()` has no effects.

`barrier::arrival_token` is an unspecified type, such that it meets the `Cpp17MoveConstructible` (Table 28), `Cpp17MoveAssignable` (Table 30), and `Cpp17Destructible` (Table 32) requirements.

```cpp
static constexpr ptrdiff_t max() noexcept;
```

Returns: The maximum expected count that the implementation supports.

```cpp
constexpr explicit barrier(ptrdiff_t expected, CompletionFunction f = CompletionFunction());
```

Preconditions: `expected >= 0` is `true` and `expected <= max()` is `true`.

Effects: Sets both the initial expected count for each barrier phase and the current expected count for the first phase to `expected`. Initializes `completion` with `std::move(f)`. Starts the first phase.

[Note 1: If `expected` is 0 this object can only be destroyed. — end note]

Throws: Any exception thrown by `CompletionFunction`'s move constructor.

```cpp
[[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
```

Preconditions: `update > 0` is `true`, and `update` is less than or equal to the expected count for the current barrier phase.

Effects: Constructs an object of type `arrival_token` that is associated with the phase synchronization point for the current phase. Then, decrements the expected count by `update`.

Synchronization: The call to `arrive` strongly happens before the start of the phase completion step for the current phase.

Returns: The constructed `arrival_token` object.

Throws: `system_error` when an exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.4.2).

[Note 2: This call can cause the completion step for the current phase to start. — end note]

```cpp
void wait(arrival_token&& arrival) const;
```

Preconditions: `arrival` is associated with the phase synchronization point for the current phase or the immediately preceding phase of the same barrier object.

Effects: Blocks at the synchronization point associated with `std::move(arrival)` until the phase completion step of the synchronization point’s phase is run.

[Note 3: If `arrival` is associated with the synchronization point for a previous phase, the call returns immediately. — end note]
21  *Throws*: `system_error` when an exception is required (32.2.2).

22  *Error conditions*: Any of the error conditions allowed for mutex types (32.5.4.2).

23  ```
void arrive_and_wait();
```

24  *Effects*: Equivalent to: `wait(arrive())`.

25  ```
void arrive_and_drop();
```

26  *Preconditions*: The expected count for the current barrier phase is greater than zero.

27  *Effects*: Decrements the initial expected count for all subsequent phases by one. Then decrements the expected count for the current phase by one.

28  *Synchronization*: The call to `arrive_and_drop` strongly happens before the start of the phase completion step for the current phase.

29  *Throws*: `system_error` when an exception is required (32.2.2).

30  *Error conditions*: Any of the error conditions allowed for mutex types (32.5.4.2).

31  `[Note 4: This call can cause the completion step for the current phase to start. — end note]`

32  32.9 Futures [futures]

33  32.9.1 Overview [futures.overview]

34  32.9 describes components that a C++ program can use to retrieve in one thread the result (value or exception) from a function that has run in the same thread or another thread.

35  [Note 1: These components are not restricted to multi-threaded programs but can be useful in single-threaded programs as well. — end note]`

36  32.9.2 Header `<future>` synopsis [future.syn]

37  ```
namespace std {
enum class future_errc {
  broken_promise = implementation-defined,
  future_already_retrieved = implementation-defined,
  promise_already_satisfied = implementation-defined,
  no_state = implementation-defined
};
enum class launch : unspecified {
  async = unspecified,
  deferred = unspecified,
  implementation-defined
};
enum class future_status {
  ready,
  timeout,
  deferred
};

template<typename T> struct is_error_code_enum<future_errc> : public true_type {};
error_code make_error_code(future_errc e) noexcept;
error_condition make_error_condition(future_errc e) noexcept;

const error_category& future_category() noexcept;

class future_error;

template<class R> class promise;
template<class R> class promise<R&>;
template<> class promise<void>;

template<class R>
  void swap(promise<R>& x, promise<R>& y) noexcept;
```
template<class R, class Alloc>
    struct uses_allocator<promise<R>, Alloc>;

template<class R> class future;
template<class R> class future<R&>;
template<> class future<void>;

template<class R> class shared_future;
template<class R> class shared_future<R&>;
template<> class shared_future<void>;

template<class> class packaged_task;
    // not defined
template<class R, class... ArgTypes>
    class packaged_task<R(ArgTypes...)>;

template<class R, class... ArgTypes>
    void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&) noexcept;

template<class F, class... Args>
    [[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
        async(F&& f, Args&&... args);

    template<class F, class... Args>
    [[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
        async(launch policy, F&& f, Args&&... args);

1 The enum type launch is a bitmask type (16.3.3.3.4) with elements launch::async and launch::deferred.

[Note 1: Implementations can provide bitmasks to specify restrictions on task interaction by functions launched by
async() applicable to a corresponding subset of available launch policies. Implementations can extend the behavior
of the first overload of async() by adding their extensions to the launch policy under the “as if” rule. — end note]

2 The enum values of future_errc are distinct and not zero.

32.9.3 Error handling [futures.errors]

const error_category& future_category() noexcept;

1 Returns: A reference to an object of a type derived from class error_category.

The object’s default_error_condition and equivalent virtual functions shall behave as specified for
the class error_category. The object’s name virtual function returns a pointer to the string "future".

error_code make_error_code(future_errc e) noexcept;

1 Returns: error_code(static_cast<int>(e), future_category()).

error_condition make_error_condition(future_errc e) noexcept;

1 Returns: error_condition(static_cast<int>(e), future_category()).

32.9.4 Class future_error [futures.future.error]

namespace std {
    class future_error : public logic_error {
        public:
            explicit future_error(future_errc e);

            const error_code& code() const noexcept;
            const char* what() const noexcept;

        private:
            error_code ec_; // exposition only
    };

    explicit future_error(future_errc e);

1 Effects: Initializes ec_ with make_error_code(e).
const error_code& code() const noexcept;

Returns: ec_

const char* what() const noexcept;

Returns: An NTBS incorporating code().message().

32.9.5 Shared state [futures.state]

Many of the classes introduced in subclause 32.9 use some state to communicate results. This shared state consists of some state information and some (possibly not yet evaluated) result, which can be a (possibly void) value or an exception.

[Note 1: Futures, promises, and tasks defined in this Clause reference such shared state. — end note]

An asynchronous return object is an object that reads results from a shared state. A waiting function of an asynchronous return object is one that potentially blocks to wait for the shared state to be made ready. If a waiting function can return before the state is made ready because of a timeout (32.2.5), then it is a timed waiting function, otherwise it is a non-timed waiting function.

An asynchronous provider is an object that provides a result to a shared state. The result of a shared state is set by respective functions on the asynchronous provider.

[Note 3: Such as promises or tasks. — end note]

The means of setting the result of a shared state is specified in the description of those classes and functions that create such a state object.

When an asynchronous return object or an asynchronous provider is said to release its shared state, it means:

(5.1) if the return object or provider holds the last reference to its shared state, the shared state is destroyed; and

(5.2) the return object or provider gives up its reference to its shared state; and

(5.3) these actions will not block for the shared state to become ready, except that it may block if all of the following are true: the shared state was created by a call to std::async, the shared state is not yet ready, and this was the last reference to the shared state.

When an asynchronous provider is said to make its shared state ready, it means:

(6.1) first, the provider marks its shared state as ready; and

(6.2) second, the provider unblocks any execution agents waiting for its shared state to become ready.

When an asynchronous provider is said to abandon its shared state, it means:

(7.1) first, if that state is not ready, the provider

(7.1.1) stores an exception object of type future_error with an error condition of broken_promise within its shared state; and then

(7.1.2) makes its shared state ready;

(7.2) second, the provider releases its shared state.

A shared state is ready only if it holds a value or an exception ready for retrieval. Waiting for a shared state to become ready may invoke code to compute the result on the waiting thread if so specified in the description of the class or function that creates the state object.

Calls to functions that successfully set the stored result of a shared state synchronize with (6.9.2) calls to functions successfully detecting the ready state resulting from that setting. The storage of the result (whether normal or exceptional) into the shared state synchronizes with (6.9.2) the successful return from a call to a waiting function on the shared state.

Some functions (e.g., promise::set_value_at_thread_exit) delay making the shared state ready until the calling thread exits. The destruction of each of that thread’s objects with thread storage duration (6.7.5.3) is sequenced before making that shared state ready.

Access to the result of the same shared state may conflict (6.9.2).
[Note 4: This explicitly specifies that the result of the shared state is visible in the objects that reference this state in the sense of data race avoidance (16.4.6.10). For example, concurrent accesses through references returned by shared_future::get() (32.9.8) must either use read-only operations or provide additional synchronization. — end note]

### 32.9.6 Class template promise

```cpp
namespace std {
    template<class R>
    class promise {
    public:
        promise();
        template<class Allocator>
        promise(allocator_arg_t, const Allocator& a);
        promise(promise&& rhs) noexcept;
        promise(const promise&) = delete;
        ~promise();

        // assignment
        promise& operator=(promise&& rhs) noexcept;
        promise& operator=(const promise&) = delete;
        void swap(promise& other) noexcept;

        // retrieving the result
        future<R> get_future();

        // setting the result
        void set_value(see below);
        void set_exception(exception_ptr p);

        // setting the result with deferred notification
        void set_value_at_thread_exit(see below);
        void set_exception_at_thread_exit(exception_ptr p);
    }

    template<class R>
    void swap(promise<R>& x, promise<R>& y) noexcept;

    template<class R, class Alloc>
    struct uses_allocator<promise<R>, Alloc> : true_type { }
};
```

1. The implementation provides the template `promise` and two specializations, `promise<R&>` and `promise<void>`. These differ only in the argument type of the member functions `set_value` and `set_value_at_thread_exit`, as set out in their descriptions, below.

2. The `set_value`, `set_exception`, `set_value_at_thread_exit`, and `set_exception_at_thread_exit` member functions behave as though they acquire a single mutex associated with the promise object while updating the promise object.

3. **Preconditions:** `Alloc` meets the `Cpp17Allocator` requirements (Table 36).

   ```cpp
   promise();
   template<class Allocator>
   promise(allocator_arg_t, const Allocator& a);
   Effects: Creates a shared state. The second constructor uses the allocator `a` to allocate memory for the shared state.
   ```

   ```cpp
   promise(promise&& rhs) noexcept;
   Effects: Transfers ownership of the shared state of `rhs` (if any) to the newly-constructed object.
   ```

   **Postconditions:** `rhs` has no shared state.
~promise();

Effects: Abandons any shared state (32.9.5).

promise& operator=(promise&& rhs) noexcept;

Effects: Abandons any shared state (32.9.5) and then as if promise(std::move(rhs)).swap(*this).

Returns: *this.

void swap(promise& other) noexcept;

Effects: Exchanges the shared state of *this and other.

Postconditions: *this has the shared state (if any) that other had prior to the call to swap. other has the shared state (if any) that *this had prior to the call to swap.

future<R> get_future();

Returns: A future<R> object with the same shared state as *this.

Synchronization: Calls to this function do not introduce data races (6.9.2) with calls to set_value, set_exception, set_value_at_thread_exit, or set_exception_at_thread_exit.

[Note 1: Such calls need not synchronize with each other. — end note]

Throws: future_error if *this has no shared state or if get_future has already been called on a promise with the same shared state as *this.

Error conditions:

(15.1) — future_already_retrieved if get_future has already been called on a promise with the same shared state as *this.

(15.2) — no_state if *this has no shared state.

void promise::set_value(const R& r);
void promise::set_value(R&& r);
void promise<R&>::set_value(R& r);
void promise<void>::set_value();

Effects: Atomically stores the value r in the shared state and makes that state ready (32.9.5).

Throws:

(17.1) — future_error if its shared state already has a stored value or exception, or

(17.2) — for the first version, any exception thrown by the constructor selected to copy an object of R, or

(17.3) — for the second version, any exception thrown by the constructor selected to move an object of R.

Error conditions:

(18.1) — promise_already_satisfied if its shared state already has a stored value or exception.

(18.2) — no_state if *this has no shared state.

void set_exception(exception_ptr p);

Preconditions: p is not null.

Effects: Atomically stores the exception pointer p in the shared state and makes that state ready (32.9.5).

Throws: future_error if its shared state already has a stored value or exception.

Error conditions:

(22.1) — promise_already_satisfied if its shared state already has a stored value or exception.

(22.2) — no_state if *this has no shared state.

void promise::set_value_at_thread_exit(const R& r);
void promise::set_value_at_thread_exit(R&& r);
void promise<R&>::set_value_at_thread_exit(R& r);
void promise<void>::set_value_at_thread_exit();

Effects: Stores the value r in the shared state without making that state ready immediately. Schedules that state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.
24 Throws:
(24.1) — future_error if its shared state already has a stored value or exception, or
(24.2) — for the first version, any exception thrown by the constructor selected to copy an object of R, or
(24.3) — for the second version, any exception thrown by the constructor selected to move an object of R.

25 Error conditions:
(25.1) — promise_already_satisfied if its shared state already has a stored value or exception.
(25.2) — no_state if *this has no shared state.

void set_exception_at_thread_exit(exception_ptr p);

26 Preconditions: p is not null.

27 Effects: Stores the exception pointer p in the shared state without making that state ready immediately.
Schedules that state to be made ready when the current thread exits, after all objects of thread storage
duration associated with the current thread have been destroyed.

28 Throws: future_error if an error condition occurs.

29 Error conditions:
(29.1) — promise_already_satisfied if its shared state already has a stored value or exception.
(29.2) — no_state if *this has no shared state.

template<class R>
void swap(promise<R>& x, promise<R>& y) noexcept;

30 Effects: As if by x.swap(y).

32.9.7 Class template future

The class template future defines a type for asynchronous return objects which do not share their shared
state with other asynchronous return objects. A default-constructed future object has no shared state. A
future object with shared state can be created by functions on asynchronous providers (32.9.5) or by the
move constructor and shares its shared state with the original asynchronous provider. The result (value or
exception) of a future object can be set by calling a respective function on an object that shares the same
shared state.

[Note 1: Member functions of future do not synchronize with themselves or with member functions of shared_future.
— end note]

The effect of calling any member function other than the destructor, the move-assignment operator, share,
or valid on a future object for which valid() == false is undefined.

[Note 2: It is valid to move from a future object for which valid() == false. — end note]

Recommended practice: Implementations should detect this case and throw an object of type future_error
with an error condition of future_errc::no_state.

namespace std {

template<class R>
class future {

public:
future() noexcept;
future(future&&) noexcept;
future(const future&) = delete;
~future();
future& operator=(const future&) = delete;
future& operator=(future&&) noexcept;
shared_future<R> share() noexcept;

// retrieving the value
see below get();

// functions to check state
bool valid() const noexcept;

§ 32.9.7 1621
The implementation provides the template `future` and two specializations, `future<R&>` and `future<void>`. These differ only in the return type and return value of the member function `get`, as set out in its description, below.

**future() noexcept;**

*Effects:* The object does not refer to a shared state.

*Postconditions:* \( \text{valid()} = \text{false} \).

**future(future&& rhs) noexcept;**

*Effects:* Move constructs a `future` object that refers to the shared state that was originally referred to by `rhs` (if any).

*Postconditions:*
- \( \text{valid()} \) returns the same value as `rhs.valid()` prior to the constructor invocation.
- \( \text{rhs.valid()} = \text{false} \).

-\( \text{future();} \)

*Effects:*
- \( \text{valid()} \) returns the same value as `valid()` prior to the assignment.
- \( \text{rhs.valid()} = \text{false} \).

**shared_future<R> share() noexcept;**

*Postconditions:* \( \text{valid()} = \text{false} \).

*Returns:* `shared_future<R>(std::move(*this))`.

> \[Note 3: As described above, the template and its two required specializations differ only in the return type and return value of the member function `get`. — end note\]
Throws: The stored exception, if an exception was stored in the shared state.

    bool valid() const noexcept;

Returns: true only if *this refers to a shared state.

void wait() const;

Effects: Blocks until the shared state is ready.

template<class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;

Effects: None if the shared state contains a deferred function (32.9.9), otherwise blocks until the shared state is ready or until the relative timeout (32.2.4) specified by rel_time has expired.

Returns:
(22.1) — future_status::deferred if the shared state contains a deferred function.
(22.2) — future_status::ready if the shared state is ready.
(22.3) — future_status::timeout if the function is returning because the relative timeout (32.2.4) specified by rel_time has expired.

Throws: timeout-related exceptions (32.2.4).

template<class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;

Effects: None if the shared state contains a deferred function (32.9.9), otherwise blocks until the shared state is ready or until the absolute timeout (32.2.4) specified by abs_time has expired.

Returns:
(25.1) — future_status::deferred if the shared state contains a deferred function.
(25.2) — future_status::ready if the shared state is ready.
(25.3) — future_status::timeout if the function is returning because the absolute timeout (32.2.4) specified by abs_time has expired.

Throws: timeout-related exceptions (32.2.4).

32.9.8 Class template shared_future

The class template shared_future defines a type for asynchronous return objects which may share their shared state with other asynchronous return objects. A default-constructed shared_future object has no shared state. A shared_future object with shared state can be created by conversion from a future object and shares its shared state with the original asynchronous provider (32.9.5) of the shared state. The result (value or exception) of a shared_future object can be set by calling a respective function on an object that shares the same shared state.

[Note 1: Member functions of shared_future do not synchronize with themselves, but they synchronize with the shared state. — end note]

The effect of calling any member function other than the destructor, the move-assignment operator, the copy-assignment operator, or valid() on a shared_future object for which valid() == false is undefined. [Note 2: It is valid to copy or move from a shared_future object for which valid() is false. — end note]

Recommended practice: Implementations should detect this case and throw an object of type future_error with an error condition of future_errc::no_state.

namespace std {
    template<class R>
    class shared_future {
        public:
            shared_future() noexcept;
            shared_future(const shared_future& rhs) noexcept;
            shared_future(future<R>&&) noexcept;
            shared_future(shared_future&& rhs) noexcept;
            ~shared_future();
            shared_future& operator=(const shared_future& rhs) noexcept;


§ 32.9.8 1623
shared_future& operator=(shared_future&& rhs) noexcept;

// retrieving the value
see below get() const;

// functions to check state
bool valid() const noexcept;

void wait() const;
template<class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
template<class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};

The implementation provides the template shared_future and two specializations, shared_future<R&> and
shared_future<void>. These differ only in the return type and return value of the member function get,
as set out in its description, below.

shared_future() noexcept;

Effects: The object does not refer to a shared state.

Postconditions: valid() == false.

shared_future(const shared_future& rhs) noexcept;

Effects: The object refers to the same shared state as rhs (if any).

Postconditions: valid() returns the same value as rhs.valid().

shared_future(future<R>&& rhs) noexcept;
shared_future(shared_future&& rhs) noexcept;

Effects: Move constructs a shared_future object that refers to the shared state that was originally
referred to by rhs (if any).

Postconditions:
- valid() returns the same value as rhs.valid() returned prior to the constructor invocation.
- rhs.valid() == false.

~shared_future();

Effects:
- Releases any shared state (32.9.5);
- destroys *this.

shared_future& operator=(shared_future&& rhs) noexcept;

Effects:
- Releases any shared state (32.9.5);
- move assigns the contents of rhs to *this.

Postconditions:
- valid() returns the same value as rhs.valid() returned prior to the assignment.
- rhs.valid() == false.

shared_future& operator=(const shared_future& rhs) noexcept;

Effects:
- Releases any shared state (32.9.5);
- assigns the contents of rhs to *this.

[Note 3: As a result, *this refers to the same shared state as rhs (if any). — end note]

Postconditions: valid() == rhs.valid().
const R& shared_future::get() const;
R& shared_future<R&>::get() const;
void shared_future<void>::get() const;

[Note 4: As described above, the template and its two required specializations differ only in the return type and return value of the member function get. — end note]

[Note 5: Access to a value object stored in the shared state is unsynchronized, so operations on R might introduce a data race (6.9.2). — end note]

Effects: wait()s until the shared state is ready, then retrieves the value stored in the shared state.

Returns:
(19.1) shared_future::get() returns a const reference to the value stored in the object’s shared state.
[Note 6: Access through that reference after the shared state has been destroyed produces undefined behavior; this can be avoided by not storing the reference in any storage with a greater lifetime than the shared_future object that returned the reference. — end note]
(19.2) shared_future<R&>::get() returns the reference stored as value in the object’s shared state.
(19.3) shared_future<void>::get() returns nothing.

Throws: The stored exception, if an exception was stored in the shared state.

bool valid() const noexcept;

Returns: true only if *this refers to a shared state.

void wait() const;

Effects: Blocks until the shared state is ready.

template<class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;

Effects: None if the shared state contains a deferred function (32.9.9), otherwise blocks until the shared state is ready or until the relative timeout (32.2.4) specified by rel_time has expired.

Returns:
(24.1) future_status::deferred if the shared state contains a deferred function.
(24.2) future_status::ready if the shared state is ready.
(24.3) future_status::timeout if the function is returning because the relative timeout (32.2.4) specified by rel_time has expired.

Throws: timeout-related exceptions (32.2.4).

template<class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;

Effects: None if the shared state contains a deferred function (32.9.9), otherwise blocks until the shared state is ready or until the absolute timeout (32.2.4) specified by abs_time has expired.

Returns:
(27.1) future_status::deferred if the shared state contains a deferred function.
(27.2) future_status::ready if the shared state is ready.
(27.3) future_status::timeout if the function is returning because the absolute timeout (32.2.4) specified by abs_time has expired.

Throws: timeout-related exceptions (32.2.4).

32.9.9 Function template async [futures.async]

The function template async provides a mechanism to launch a function potentially in a new thread and provides the result of the function in a future object with which it shares a shared state.

template<class F, class... Args>
[[nodosccard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
async(F&& f, Args&&... args);
template<class F, class... Args>
   [[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
async(launch policy, F&& f, Args&&... args);

Mandates: The following are all true:
   - (2.1) is_constructible_v<decay_t<F>, F>,
   - (2.2) (is_constructible_v<decay_t<Args>, Args> &&...),
   - (2.3) is_move_constructible_v<decay_t<F>>,
   - (2.4) (is_move_constructible_v<decay_t<Args>> &&...), and
   - (2.5) is_invocable_v<decay_t<F>, decay_t<Args>...>.

Preconditions: decay_t<F> and each type in decay_t<Args> meet the Cpp17MoveConstructible requirements.

Effects: The first function behaves the same as a call to the second function with a policy argument of launch::async | launch::deferred and the same arguments for F and Args. The second function creates a shared state that is associated with the returned future object. The further behavior of the policy argument depends on the policy argument as follows (if more than one of these conditions applies, the implementation may choose any of the corresponding policies):
   - (4.1) If launch::async is set in policy, calls invoke(decay-copy(std::forward<F>(f)), decay-copy(std::forward<Args>(args))...) (20.14.4, 32.4.3.3) as if in a new thread of execution represented by a thread object with the calls to decay-copy being evaluated in the thread that called async. Any return value is stored as the result in the shared state. Any exception propagated from the execution of invoke(decay-copy(std::forward<F>(f)), decay-copy(std::forward<Args>(args))...) is stored as the exceptional result in the shared state. The thread object is stored in the shared state and affects the behavior of any asynchronous return objects that reference that state.
   - (4.2) If launch::deferred is set in policy, stores decay-copy(std::forward<F>(f)) and decay-copy(std::forward<Args>(args))... in the shared state. These copies of f and args constitute a deferred function. Invocation of the deferred function evaluates invoke(std::move(g), std::move(xyz)) where g is the stored value of decay-copy(std::forward<F>(f)) and xyz is the stored copy of decay-copy(std::forward<Args>(args))... Any return value is stored as the result in the shared state. Any exception propagated from the execution of the deferred function is stored as the exceptional result in the shared state. The shared state is not made ready until the function has completed. The first call to a non-timed waiting function (32.9.5) on an asynchronous return object referring to this shared state invokes the deferred function in the thread that called the waiting function. Once evaluation of invoke(std::move(g), std::move(xyz)) begins, the function is no longer considered deferred.

Recommended practice: If this policy is specified together with other policies, such as when using a policy value of launch::async | launch::deferred, implementations should defer invocation or the selection of the policy when no more concurrency can be effectively exploited.
   - (4.3) If no value is set in the launch policy, or a value is set that is neither specified in this document nor by the implementation, the behavior is undefined.

Returns: An object of type future<invoke_result_t<decay_t<F>, decay_t<Args>...>> that refers to the shared state created by this call to async.

[Note 1: If a future obtained from async is moved outside the local scope, the future's destructor can block for the shared state to become ready. — end note]

Synchronization: The invocation of async synchronizes with the invocation of f. The completion of the function f is sequenced before the shared state is made ready.

[Note 2: These apply regardless of the provided policy argument, and even if the corresponding future object is moved to another thread. However, f might not be called at all, so its completion might never happen. — end note]

If the implementation chooses the launch::async policy,
   - (6.1) a call to a waiting function on an asynchronous return object that shares the shared state created by this async call shall block until the associated thread has completed, as if joined, or else time out (32.4.3.6);
the associated thread completion synchronizes with (6.9.2) the return from the first function that successfully detects the ready status of the shared state or with the return from the last function that releases the shared state, whichever happens first.

Throws: `system_error` if `policy == launch::async` and the implementation is unable to start a new thread, or `std::bad_alloc` if memory for the internal data structures could not be allocated.

Error conditions:

— resource_unavailable_try_again — if `policy == launch::async` and the system is unable to start a new thread.

Example 1:

```cpp
int work1(int value);
int work2(int value);
int work(int value) {
    auto handle = std::async([=]{ return work2(value); });
    int tmp = work1(value);
    return tmp + handle.get(); // #1
}
```

[Note 3: Line #1 might not result in concurrency because the `async` call uses the default policy, which might use `launch::deferred`, in which case the lambda might not be invoked until the `get()` call; in that case, `work1` and `work2` are called on the same thread and there is no concurrency. — end note]

— end example]

32.9.10 Class template `packaged_task` [futures.task]

32.9.10.1 General [futures.task.general]

The class template `packaged_task` defines a type for wrapping a function or callable object so that the return value of the function or callable object is stored in a future when it is invoked.

When the `packaged_task` object is invoked, its stored task is invoked and the result (whether normal or exceptional) stored in the shared state. Any futures that share the shared state will then be able to access the stored result.

```cpp
namespace std {
    template<class> class packaged_task; // not defined

    template<class R, class... ArgTypes>
    class packaged_task<R(ArgTypes...)> {
        public:
            // construction and destruction
            packaged_task() noexcept;
            template<class F>
                explicit packaged_task(F&& f);
            ~packaged_task();

            // no copy
            packaged_task(const packaged_task&) = delete;
            packaged_task& operator=(const packaged_task&) = delete;

            // move support
            packaged_task(packaged_task&& rhs) noexcept;
            packaged_task& operator=(packaged_task&& rhs) noexcept;
            void swap(packaged_task& other) noexcept;

            bool valid() const noexcept;

            // result retrieval
            future<R> get_future();

            // void operator()(ArgTypes...);
            void make_ready_at_thread_exit(ArgTypes...);
    }
}
```
void reset();
);

```c

template<class R, class... ArgTypes>
void swap(packaged_task<R(ArgTypes...)> & x, packaged_task<R(ArgTypes...)> & y) noexcept;
```

### 32.9.10.2 Member functions

#### packaged_task() noexcept;

*Effects:* The object has no shared state and no stored task.

*Constraints:* `remove_cvref_t<F>` is not the same type as `packaged_task<R(ArgTypes...)>`.

*Mandates:* `is_invocable_r_v<R, F&, ArgTypes...>` is true.

*Preconditions:* Invoking a copy of `f` behaves the same as invoking `f`.

*Effects:* Constructs a new `packaged_task` object with a shared state and initializes the object's stored task with `std::forward<F>(f)`.

*Throws:* Any exceptions thrown by the copy or move constructor of `f`, or `bad_alloc` if memory for the internal data structures could not be allocated.

#### packaged_task(packaged_task&& rhs) noexcept;

*Effects:* Transfers ownership of `rhs`'s shared state to *this, leaving `rhs` with no shared state. Moves the stored task from `rhs` to *this.

*Postconditions:* `rhs` has no shared state.

#### packaged_task& operator=(packaged_task&& rhs) noexcept;

*Effects:*

- (9.1) Releases any shared state (32.9.5);
- (9.2) calls `packaged_task(std::move(rhs)).swap(*this)`.

*Postconditions:* *this has no shared state.

#### ~packaged_task();

*Effects:* Abandons any shared state (32.9.5).

#### void swap(packaged_task& other) noexcept;

*Effects:* Exchanges the shared states and stored tasks of *this and other.

*Postconditions:* *this has the same shared state and stored task (if any) as other prior to the call to `swap`. other has the same shared state and stored task (if any) as *this prior to the call to `swap`.

bool valid() const noexcept;

*Returns:* `true` only if *this has a shared state.

#### future<R> get_future();

*Returns:* A `future` object that shares the same shared state as *this.

*Synchronization:* Calls to this function do not introduce data races (6.9.2) with calls to `operator()` or `make_ready_at_thread_exit`.

*[Note 1: Such calls need not synchronize with each other. — end note]*

*Throws:* A `future_error` object if an error occurs.

*Error conditions:*

- (17.1) `future_already_retrieved` if `get_future` has already been called on a `packaged_task` object with the same shared state as *this.
- (17.2) `no_state` if *this has no shared state.

§ 32.9.10.2
void operator()(ArgTypes... args);

Effects: As if by INVOKE<R>(f, t₁, t₂, ..., tₙ) (20.14.4), where f is the stored task of *this and t₁, t₂, ..., tₙ are the values in args... If the task returns normally, the return value is stored as the asynchronous result in the shared state of *this, otherwise the exception thrown by the task is stored. The shared state of *this is made ready, and any threads blocked in a function waiting for the shared state of *this to become ready are unblocked.

Throws: A future_error exception object if there is no shared state or the stored task has already been invoked.

Error conditions:

20.1 promise_already_satisfied if the stored task has already been invoked.
20.2 no_state if *this has no shared state.

void make_ready_at_thread_exit(ArgTypes... args);

Effects: As if by INVOKE<R>(f, t₁, t₂, ..., tₙ) (20.14.4), where f is the stored task and t₁, t₂, ..., tₙ are the values in args... If the task returns normally, the return value is stored as the asynchronous result in the shared state of *this, otherwise the exception thrown by the task is stored. In either case, this is done without making that state ready (32.9.5) immediately. Schedules the shared state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws: future_error if an error condition occurs.

Error conditions:

21.1 promise_already_satisfied if the stored task has already been invoked.
21.2 no_state if *this has no shared state.

void reset();

Effects: As if *this = packaged_task(std::move(f)), where f is the task stored in *this.

[Note 2: This constructs a new shared state for *this. The old state is abandoned (32.9.5). — end note]

Throws:

25.1 bad_alloc if memory for the new shared state could not be allocated.
25.2 any exception thrown by the move constructor of the task stored in the shared state.
25.3 future_error with an error condition of no_state if *this has no shared state.

32.9.10.3 Globals

template<class R, class... ArgTypes>
void swap(packaged_task<R(ArgTypes...)>& x, packaged_task<R(ArgTypes...)>& y) noexcept;

Effects: As if by x.swap(y).
Annex A  (informative)
Grammar summary

A.1 General

This summary of C++ grammar is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, the grammar described here accepts a superset of valid C++ constructs. Disambiguation rules (8.9, 9.2, 11.8) are applied to distinguish expressions from declarations. Further, access control, ambiguity, and type rules are used to weed out syntactically valid but meaningless constructs.

A.2 Keywords

New context-dependent keywords are introduced into a program by typedef (9.2.4), namespace (9.8.2), class (Clause 11), enumeration (9.7.1), and template (Clause 13) declarations.

A.3 Lexical conventions
header-name:
  < h-char-sequence >
  " q-char-sequence "

h-char-sequence:
  h-char
  h-char-sequence h-char

h-char:
  any member of the source character set except new-line and >

q-char-sequence:
  q-char
  q-char-sequence q-char

q-char:
  any member of the source character set except new-line and "

pp-number:
  digit
  . digit
  pp-number digit
  pp-number identifier-nondigit
  pp-number ' digit
  pp-number ' nondigit
  pp-number e sign
  pp-number E sign
  pp-number p sign
  pp-number P sign
  pp-number .

identifier:
  identifier-nondigit
  identifier identifier-nondigit
  identifier digit

identifier-nondigit:
  nondigit
  universal-character-name

nondigit: one of
  a b c d e f g h i j k l m
  n o p q r s t u v w x y z

  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z

digit: one of
  0 1 2 3 4 5 6 7 8 9

keyword:
  any identifier listed in Table 5

import-keyword
module-keyword
export-keyword

preprocessing-op-or-punc:
  preprocessing-operator
  operator-or-punctuator

preprocessing-operator: one of
  #   ##   %   %:

operator-or-punctuator: one of
  {   }   [   ]   (   )   ,   ...

  ?   ::   .   *   ->   ->*

  !   +   -   *   /   %   ^   &   |
  =   +=   -=   ==   /=   %=   ^=   &=   |=
  <<=   >>=   <<=   >>=   <=   >=   <=>  &<<

  <   >>
  <=  <<=
  ==  >=

  and  or  xor  not  bitand  bitor  compl
  and_eq  or_eq  xor_eq  not_eq

§ A.3  1631
literal:
  integer-literal
  character-literal
  floating-point-literal
  string-literal
  boolean-literal
  pointer-literal
  user-defined-literal

integer-literal:
  binary-literal integer-suffix
  octal-literal integer-suffix
  decimal-literal integer-suffix
  hexadecimal-literal integer-suffix

binary-literal:
  0b binary-digit
  0B binary-digit
  binary-literal 'opt binary-digit

octal-literal:
  0
  octal-literal 'opt octal-digit

decimal-literal:
  nonzero-digit
  decimal-literal 'opt digit

hexadecimal-literal:
  hexadecimal-prefix hexadecimal-digit-sequence

binary-digit: one of
  0 1

octal-digit: one of
  0 1 2 3 4 5 6 7

nonzero-digit: one of
  1 2 3 4 5 6 7 8 9

hexadecimal-prefix: one of
  0x 0X

hexadecimal-digit-sequence:
  hexadecimal-digit
  hexadecimal-digit-sequence 'opt hexadecimal-digit

hexadecimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-suffix:
  unsigned-suffix long-suffix
  unsigned-suffix long-long-suffix
  long-suffix unsigned-suffix
  long-long-suffix unsigned-suffix

unsigned-suffix: one of
  u U

long-suffix: one of
  l L

long-long-suffix: one of
  ll LL

character-literal:
  encoding-prefix 'opt c-char-sequence '

encoding-prefix: one of
  u8 u U L
c-char-sequence:
c-char
  c-char-sequence c-char
c-char:
  any member of the basic source character set except the single-quote '\', backslash \, or new-line character escape-sequence
  universal-character-name
escape-sequence:
simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence
simple-escape-sequence:
one of
  \ ' " \? \" \a \b \f \n \r \t \v
octal-escape-sequence:
  \ octal-digit
  \ octal-digit octal-digit
  \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
  \x hexadecimal-digit
  hexadecimal-escape-sequence hexadecimal-digit
floating-point-literal:
  decimal-floating-point-literal
  hexadecimal-floating-point-literal
decimal-floating-point-literal:
  fractional-constant exponent-part opt floating-point-suffix opt
  digit-sequence exponent-part floating-point-suffix opt
hexadecimal-floating-point-literal:
  hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-point-suffix opt
  hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-point-suffix opt
fractional-constant:
  digit-sequence opt . digit-sequence
  digit-sequence
hexadecimal-fractional-constant:
  hexadecimal-digit-sequence opt . hexadecimal-digit-sequence
  hexadecimal-digit-sequence
exponent-part:
  e sign opt digit-sequence
  E sign opt digit-sequence
binary-exponent-part:
  p sign opt digit-sequence
  P sign opt digit-sequence
sign:
one of
  + -
digit-sequence:
  digit
  digit-sequence opt digit
floating-point-suffix: one of
  f F L
string-literal:
  encoding-prefix opt " s-char-sequence opt "
  encoding-prefix opt R raw-string
s-char-sequence:
  s-char
  s-char-sequence s-char
s-char:
  any member of the basic source character set except the double-quote ",", backslash \, or new-line character
  escape-sequence
  universal-character-name

raw-string:
  " d-char-sequence_{opt} ( r-char-sequence_{opt} ) d-char-sequence_{opt} "

r-char-sequence:
  r-char
  r-char-sequence r-char

r-char:
  any member of the source character set, except a right parenthesis ) followed by
  the initial d-char-sequence (which may be empty) followed by a double quote ".

d-char-sequence:
  d-char
  d-char-sequence d-char

d-char:
  any member of the basic source character set except:
    space, the left parenthesis (, the right parenthesis ), the backslash \, and the control characters
    representing horizontal tab, vertical tab, form feed, and newline.

boolean-literal:
  false
  true

pointer-literal:
  nullptr

user-defined-literal:
  user-defined-integer-literal
  user-defined-floating-point-literal
  user-defined-string-literal
  user-defined-character-literal

user-defined-integer-literal:
  decimal-literal ud-suffix
  octal-literal ud-suffix
  hexadecimal-literal ud-suffix
  binary-literal ud-suffix

user-defined-floating-point-literal:
  fractional-constant exponent-part_{opt} ud-suffix
  digit-sequence exponent-part ud-suffix
  hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part ud-suffix
  hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part ud-suffix

user-defined-string-literal:
  string-literal ud-suffix

user-defined-character-literal:
  character-literal ud-suffix

ud-suffix:
  identifier

A.4 Basics

A.5 Expressions

§ A.5 1634
primary-expression:
  literal
  this
  ( expression )
  id-expression
  lambda-expression
  fold-expression
  requires-expression

id-expression:
  unqualified-id
  qualified-id

unqualified-id:
  identifier
  operator-function-id
  conversion-function-id
  literal-operator-id
  ~ type-name
  ~ decltype-specifier
  template-id

qualified-id:
  nested-name-specifier template_opt unqualified-id

nested-name-specifier:
  ::
    type-name ::
    namespace-name ::
    decltype-specifier ::
    nested-name-specifier identifier ::
    nested-name-specifier template_opt simple-template-id ::

lambda-expression:
  lambda-introducer lambda-declarator_opt compound-statement
  lambda-introducer < template-parameter-list > requires-clause_opt lambda-declarator_opt compound-statement

lambda-introducer:
  [ lambda-capture_opt ]

lambda-declarator:
  ( parameter-declaration-clause ) decl-specifier-seq_opt
    noexcept-specifier_opt attribute-specifier-seq_opt trailing-return-type_opt requires-clause_opt

lambda-capture:
  capture-default
  capture-list
  capture-default , capture-list

capture-default:
  & =

capture-list:
  capture
  capture-list , capture

capture:
  simple-capture
  init-capture

simple-capture:
  identifier . . . opt
  & identifier . . . opt
  this
  * this

init-capture:
  . . . opt identifier initializer
  & . . . opt identifier initializer
fold-expression:
  ( cast-expression fold-operator ... )
  ( ... fold-operator cast-expression )
  ( cast-expression fold-operator ... fold-operator cast-expression )

fold-operator: one of
  + - * / % ^ & | << >>
  += -= *= /= %= ^= &= |= <<< >>= =
  == != < > <= >= && || , .* -*

requires-expression:
  requires requirement-parameter-listopt requirement-body

requirement-parameter-list:
  ( parameter-declaration-clauseopt )

requirement-body:
  { requirement-seq }

requirement-seq:
  requirement
  requirement-seq requirement

requirement:
  simple-requirement
  type-requirement
  compound-requirement
  nested-requirement

simple-requirement:
  expression ;

type-requirement:
  typename nested-name-specifieropt type-name ;

compound-requirement:
  { expression } noexceptopt return-type-requirementopt ;

return-type-requirement:
  -> type-constraint

nested-requirement:
  requires constraint-expression ;

postfix-expression:
  primary-expression
  postfix-expression [ expr-or-braced-init-list ]
  postfix-expression ( expression-listopt )
  simple-type-specifier ( expression-listopt )
  typename-specifier ( expression-listopt )
  simple-type-specifier braced-init-list
  typename-specifier braced-init-list
  postfix-expression . templateopt id-expression
  postfix-expression -> templateopt id-expression
  postfix-expression ++
  postfix-expression --
  dynamic_cast < type-id > ( expression )
  static_cast < type-id > ( expression )
  reinterpret_cast < type-id > ( expression )
  const_cast < type-id > ( expression )
  typeid ( expression )
  typeid ( type-id )

expression-list:
  initializer-list
unary-expression:
  postfix-expression
  unary-operator cast-expression
  ++ cast-expression
  -- cast-expression
  await-expression
  sizeof unary-expression
  sizeof ( type-id )
  sizeof ... ( identifier )
  alignof ( type-id )
  noexcept-expression
  new-expression
  delete-expression

unary-operator: one of
  * & + - ! ~

await-expression:
  co_await cast-expression

noexcept-expression:
  noexcept ( expression )

new-expression:
  :: opt new new-placement_opt new-type-id new-initializer_opt
  :: opt new new-placement_opt ( type-id ) new-initializer_opt

new-placement:
  ( expression-list )

new-type-id:
  type-specifier-seq new-declarator_opt

new-declarator:
  ptr-operator new-declarator_opt
  noptr-new-declarator

noptr-new-declarator:
  [ expression_opt ] attribute-specifier-seq_opt
  nopt-new-declarator [ constant-expression ] attribute-specifier-seq_opt

new-initializer:
  ( expression-list_opt )
  braced-init-list

delete-expression:
  :: opt delete cast-expression
  :: opt delete [] cast-expression

cast-expression:
  unary-expression
  ( type-id ) cast-expression

pm-expression:
  cast-expression
  pm-expression . * cast-expression
  pm-expression . > > cast-expression

multiplicative-expression:
  pm-expression
  multiplicative-expression * pm-expression
  multiplicative-expression / pm-expression
  multiplicative-expression % pm-expression

additive-expression:
  multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression - multiplicative-expression

shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression >> additive-expression
A.6 Statements

[gram.stmt]
statement:
  labeled-statement
  attribute-specifier-seq_opt expression-statement
  attribute-specifier-seq_opt compound-statement
  attribute-specifier-seq_opt selection-statement
  attribute-specifier-seq_opt iteration-statement
  attribute-specifier-seq_opt jump-statement
  declaration-statement
  attribute-specifier-seq_opt try-block

init-statement:
  expression-statement
  simple-declaration

condition:
  expression
  attribute-specifier-seq_opt decl-specifier-seq declarator brace-or-equal-initializer

labeled-statement:
  attribute-specifier-seq_opt identifier : statement
  attribute-specifier-seq_opt case constant-expression : statement
  attribute-specifier-seq_opt default : statement

expression-statement:
  expression_opt ;

compound-statement:
  { statement-seq_opt }

statement-seq:
  statement
  statement-seq statement

selection-statement:
  if constexpr_opt ( init-statement_opt condition ) statement
  if constexpr_opt ( init-statement_opt condition ) statement else statement
  switch ( init-statement_opt condition ) statement

iteration-statement:
  while ( condition ) statement
  do statement while ( expression ) ;
  for ( init-statement_opt condition_opt ; expression_opt ) statement
  for ( init-statement_opt for-range-declaration : for-range-initializer ) statement

for-range-declaration:
  attribute-specifier-seq_opt decl-specifier-seq declarator
  attribute-specifier-seq_opt decl-specifier-seq ref-qualifier_opt [ identifier-list ]

for-range-initializer:
  expr-or-braced-init-list

jump-statement:
  break ;
  continue ;
  return expr-or-braced-init-list_opt ;
  coroutine-return-statement
  goto identifier ;

coroutine-return-statement:
  co_return expr-or-braced-init-list_opt ;

declaration-statement:
  block-declaration

A.7 Declarations [gram.dcl]
declaration-seq:
  declaration
  declaration-seq declaration
declaration:
  block-declaration
  nodeclspec-function-declaration
  function-definition
  template-declaration
  deduction-guide
  explicit-instantiation
  explicit-specialization
  export-declaration
  linkage-specification
  namespace-definition
  empty-declaration
  attribute-declaration
  module-import-declaration

block-declaration:
  simple-declaration
  asm-declaration
  namespace-alias-definition
  using-declaration
  using-enum-declaration
  using-directive
  static_assert-declaration
  alias-declaration
  opaque-enum-declaration

nodeclspec-function-declaration:
  attribute-specifier-seq_opt declarator;

alias-declaration:
  using identifier attribute-specifier-seq_opt = defining-type-id;

simple-declaration:
  decl-specifier-seq init-declarator-list_opt ;
  attribute-specifier-seq decl-specifier-seq init-declarator-list ;
  attribute-specifier-seq_opt decl-specifier-seq ref-qualifier_opt [ identifier-list ] initializer;

static_assert-declaration:
  static_assert ( constant-expression );
  static_assert ( constant-expression , string-literal );

empty-declaration:
  ;

attribute-declaration:
  attribute-specifier-seq ;

decl-specifier:
  storage-class-specifier
  defining-type-specifier
  function-specifier
  friend
typedef
constexpr
constexpr
consteval
constinit
inline

decl-specifier-seq:
  decl-specifier attribute-specifier-seq_opt
decl-specifier decl-specifier-seq

storage-class-specifier:
  static
  thread_local
  extern
  mutable
function-specifier:
  virtual
  explicit-specifier

explicit-specifier:
  explicit ( constant-expression )
  explicit

typedef-name:
  identifier
  simple-template-id

type-specifier:
  simple-type-specifier
  elaborated-type-specifier
  typename-specifier
  cv-qualifier

type-specifier-seq:
  type-specifier attribute-specifier-seq_opt
  type-specifier type-specifier-seq

defining-type-specifier:
  type-specifier
  class-specifier
  enum-specifier

defining-type-specifier-seq:
  defining-type-specifier attribute-specifier-seq_opt
  defining-type-specifier defining-type-specifier-seq

simple-type-specifier:
  nested-name-specifier_opt type-name
  nested-name-specifier template simple-template-id
  decltype-specifier
  placeholder-type-specifier
  nested-name-specifier_opt template-name
  char
  char8_t
  char16_t
  char32_t
  wchar_t
  bool
  short
  int
  long
  signed
  unsigned
  float
  double
  void

type-name:
  class-name
  enum-name
  typedef-name

elaborated-type-specifier:
  class-key attribute-specifier-seq_opt nested-name-specifier_opt identifier
  class-key simple-template-id
  class-key nested-name-specifier template_opt simple-template-id
  elaborated-enum-specifier

elaborated-enum-specifier:
  enum nested-name-specifier_opt identifier

decltype-specifier:
  decltype ( expression )
placeholder-type-specifier:
  type-constraint_opt auto
type-constraint_opt decltype ( auto )

init-declarator-list:
  init-declarator
  init-declarator-list , init-declarator

init-declarator:
  declarator initializer_opt
  declarator requires-clause
declarator:
  ptr-declarator
  noptr-declarator parameters-and-qualifiers trailing-return-type

ptr-declarator:
  noptr-declarator
  ptr-operator ptr-declarator

noptr-declarator:
  declarator-id attribute-specifier-seq_opt
  noptr-declarator parameters-and-qualifiers
  noptr-declarator [ constant-expression_opt ] attribute-specifier-seq_opt
  ( ptr-declarator )

parameters-and-qualifiers:
  ( parameter-declaration-clause ) cv-qualifier-seq_opt
    ref-qualifier_opt noexcept-specifier_opt attribute-specifier-seq_opt

trailing-return-type:
  -> type-id

ptr-operator:
  * attribute-specifier-seq_opt cv-qualifier-seq_opt
  & attribute-specifier-seq_opt
  && attribute-specifier-seq_opt
  nested-name-specifier * attribute-specifier-seq_opt cv-qualifier-seq_opt
cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq_opt
cv-qualifier:
  const
  volatile
ref-qualifier:
  &
  &&
declarator-id:
  ..._opt id-expression
type-id:
  type-specifier-seq abstract-declarator_opt
defining-type-id:
  defining-type-specifier-seq abstract-declarator_opt
abstract-declarator:
  ptr-abstract-declarator
  noptr-abstract-declarator_opt parameters-and-qualifiers trailing-return-type
  abstract-pack-declarator
ptr-abstract-declarator:
  noptr-abstract-declarator
  ptr-operator ptr-abstract-declarator_opt

noptr-abstract-declarator:
  noptr-abstract-declarator_opt parameters-and-qualifiers
  noptr-abstract-declarator_opt [ constant-expression_opt ] attribute-specifier-seq_opt
  ( ptr-abstract-declarator )
abstract-pack-declarator:
  noptr-abstract-pack-declarator
  ptr-operator abstract-pack-declarator

noptr-abstract-pack-declarator:
  noptr-abstract-pack-declarator parameters-and-qualifiers
  noptr-abstract-pack-declarator [ constant-expressionopt ] attribute-specifier-seqopt

parameter-declaration-clause:
  parameter-declaration-listopt ...opt
  parameter-declaration-list , ...

parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list , parameter-declaration

parameter-declaration:
  attribute-specifier-seqopt decl-specifier-seq declarator
  attribute-specifier-seqopt decl-specifier-seq declarator = initializer-clause
  attribute-specifier-seqopt decl-specifier-seq abstract-declaratoropt
  attribute-specifier-seqopt decl-specifier-seq abstract-declaratoropt = initializer-clause

initializer:
  brace-or-equal-initializer
  ( expression-list )

brace-or-equal-initializer:
  = initializer-clause
  braced-init-list

initializer-clause:
  assignment-expression
  braced-init-list

braced-init-list:
  { initializer-list ,opt }
  { designated-initializer-list ,opt }
  { }

initializer-list:
  initializer-clause ...opt
  initializer-list , initializer-clause ...opt

designated-initializer-list:
  designated-initializer-clause
  designated-initializer-list , designated-initializer-clause

designated-initializer-clause:
  designator brace-or-equal-initializer

designator:
  . identifier

expr-or-braced-init-list:
  expression
  braced-init-list

function-definition:
  attribute-specifier-seqopt decl-specifier-seqopt declarator virt-specifier-seqopt function-body
  attribute-specifier-seqopt decl-specifier-seqopt declarator requires-clause function-body

function-body:
  ctor-initializeropt compound-statement
  function-try-block
  = default ;
  = delete ;

decorator:
  identifier

decorator:
  enum-head \{ enumerator-listopt \}
  enum-head \{ enumerator-list , \}
enum-head:
  enum-key attribute-specifier-seq\opt enum-head-name\opt enum-base\opt
enum-head-name:
  nested-name-specifier\opt identifier
opaque_enum-declaration:
  enum-key attribute-specifier-seq\opt enum-head-name enum-base\opt ;
enum-key:
  enum
class
struct
enum-base:
  : type-specifier-seq
enumerator-list:
  enumerator-definition
  enumerator-list \opt enumerator-definition
enumerator-definition:
  enumerator
  enumerator = constant-expression
enumerator:
  identifier attribute-specifier-seq\opt
using_enum-declaration:
  using elaborated_enum-specifier ;
namespace-name:
  identifier
  namespace-alias
namespace-definition:
  named_namespace-definition
  unnamed_namespace-definition
  nested_namespace-definition
named_namespace-definition:
  inline\opt namespace attribute-specifier-seq\opt identifier { namespace-body }
unnamed_namespace-definition:
  inline\opt namespace attribute-specifier-seq\opt { namespace-body }
nested_namespace-definition:
  namespace enclosing_namespace-specifier :: inline\opt identifier { namespace-body }
enclosing_namespace-specifier:
  identifier
  enclosing_namespace-specifier :: inline\opt identifier
namespace-body:
  declaration-seq\opt
namespace-alias:
  identifier
namespace_alias-definition:
  namespace identifier = qualified_namespace-specifier ;
qualified_namespace-specifier:
  nested_name-specifier\opt namespace-name
using-directive:
  attribute_specifier-seq\opt using namespace nested_name-specifier\opt namespace-name ;
using-declaration:
  using using_declarator-list ;
using_declarator-list:
  using_declarator \opt
  using_declarator-list \opt using_declarator \opt
using_declarator:
  typename\opt nested_name-specifier unqualified_id
A.8 Modules

module-declaration:
    export-keyword_opt module-keyword module-name module-partition_opt attribute-specifier-seq_opt ;

module-name:
    module-name-qualifier_opt identifier

module-partition:
    : module-name-qualifier_opt identifier

module-name-qualifier:
    identifier . 
    module-name-qualifier identifier .

export-declaration:
    export declaration
    export { declaration-seq_opt }
    export-keyword module-import-declaration
module-import-declaration:
import-keyword module-name attribute-specifier-seq\opt ;
import-keyword module-partition attribute-specifier-seq\opt ;
import-keyword header-name attribute-specifier-seq\opt ;

global-module-fragment:
module-keyword ; declaration-seq\opt

private-module-fragment:
module-keyword : private ; declaration-seq\opt

A.9 Classes

class-name:
identifier
simple-template-id

class-specifier:
class-head { member-specification\opt }

class-head:
class-key attribute-specifier-seq\opt class-head-name class-virt-specifier\opt base-clause\opt
class-key attribute-specifier-seq\opt base-clause\opt

class-head-name:
nested-name-specifier\opt class-name

class-virt-specifier:
final
class-key:
class
struct
union

member-specification:
member-declaration member-specification\opt
access-specifier : member-specification\opt

member-declaration:
attribute-specifier-seq\opt decl-specifier-seq\opt member-declarator-list\opt ;
function-definition
using-declaration
using-enum-declaration
static_assert-declaration
template-declaration
explicit-specialization
deduction-guide
alias-declaration
opaque-enum-declaration
empty-declaration

member-declarator-list:
member-declarator
member-declarator-list , member-declarator

member-declarator:
declarator virt-specifier-seq\opt pure-specifier\opt
declarator requires-clause
declarator brace-or-equal-initializer\opt
identifier\opt attribute-specifier-seq\opt : constant-expression brace-or-equal-initializer\opt

virt-specifier-seq:
virt-specifier
virt-specifier-seq virt-specifier

virt-specifier:
override
final

pure-specifier:
= 0
A.10 Overloading

operator-function-id:
  operator operator

operator: one of
  new       delete     new[]     delete[]    co.await ()     []     ->     -*
  ~     !    +      -      *      /      %     ^     &
  |    &=    +=    -=    **    /=    %=    ^=    &=
  |=   ==    !==    <     >     <=    >=    <=>   &&
  | |  <<    >>    <<=    >>=    ++    --    ,

literal-operator-id:
  operator string-literal identifier
  operator user-defined-string-literal

A.11 Templates

template-declaration:
  template-head declaration
  template-head concept-definition

template-head:
  template < template-parameter-list > requires-clause_opt
template-parameter-list:
  template-parameter
  template-parameter-list , template-parameter

requires-clause:
  requires constraint-logical-or-expression

constraint-logical-or-expression:
  constraint-logical-and-expression
  constraint-logical-or-expression | constraint-logical-and-expression

constraint-logical-and-expression:
  primary-expression
  constraint-logical-and-expression && primary-expression

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
  type-parameter-key ...opt identifieropt
  type-parameter-key identifieropt = type-id
  type-constraint ...opt identifieropt
  type-constraint identifieropt = type-id
  template-head type-parameter-key ...opt identifieropt
  template-head type-parameter-key identifieropt = id-expression

type-parameter-key:
  class
typename

type-constraint:
  nested-name-specifieropt concept-name
  nested-name-specifieropt concept-name < template-argument-listopt >

simple-template-id:
  template-name < template-argument-listopt >

template-id:
  simple-template-id
  operator-function-id < template-argument-listopt >
  literal-operator-id < template-argument-listopt >

template-name:
  identifier
template-argument-list:
  template-argument ...opt
template-argument-list , template-argument ...opt
template-argument:
  constant-expression
type-id
  id-expression
constraint-expression:
  logical-or-expression
deduction-guide:
  explicit-specifieropt template-name ( parameter-declaration-clause ) -> simple-template-id ;
concept-definition:
  concept concept-name = constraint-expression ;
concept-name:
  identifier
typename-specifier:
  typename nested-name-specifier identifier
typename nested-name-specifier templateopt simple-template-id
explicit-instantiation:
  externopt template declaration
explicit-specialization:
  template <> declaration

A.12 Exception handling [gram.except]

try-block:
  try compound-statement handler-seq
function-try-block:
  try ctor-initializer\opt compound-statement handler-seq
handler-seq:
  handler handler-seq\opt
handler:
  catch ( exception-declaration ) compound-statement
exception-declaration:
  attribute-specifier-seq\opt type-specifier-seq declarator
  attribute-specifier-seq\opt type-specifier-seq abstract-declarator\opt
  ... 
noexcept-specifier:
  noexcept ( constant-expression )
  noexcept

A.13 Preprocessing directives [gram.cpp]

preprocessing-file:
  group\opt
  module-file
module-file:
  pp-global-module-fragment\opt pp-module group\opt pp-private-module-fragment\opt
pp-global-module-fragment:
  module ; new-line group\opt
pp-private-module-fragment:
  module : private ; new-line group\opt
group:
  group-part
  group group-part
group-part:
  control-line
  if-section
  text-line
  # conditionally-supported-directive
control-line:
  # include pp-tokens new-line
  pp-import
  # define identifier replacement-list new-line
  # define identifier (paren identifier-list\opt) replacement-list new-line
  # define identifier (paren ... ) replacement-list new-line
  # define identifier (paren identifier-list , ... ) replacement-list new-line
  # undef identifier new-line
  # line pp-tokens new-line
  # error pp-tokens\opt new-line
  # pragma pp-tokens\opt new-line
  # new-line
if-section:
  if-group elif-groups\opt else-group\opt endif-line
if-group:
  # if constant-expression new-line group\opt
  # ifdef identifier new-line group\opt
  # ifndef identifier new-line group\opt
  # ifndef identifier new-line group\opt
  # ifndef identifier new-line group\opt

§ A.13
elif-groups:
  elif-group
  elif-groups elif-group

elif-group:
  # elif constant-expression new-line group_opt

else-group:
  # else new-line group_opt

endif-line:
  # endif new-line

text-line:
  pp-tokens_opt new-line

conditionally-supported-directive:
  pp-tokens new-line

lparen:
  a ( character not immediately preceded by whitespace

identifier-list:
  identifier
  identifier-list , identifier

replacement-list:
  pp-tokens_opt

pp-tokens:
  preprocessing-token
  pp-tokens preprocessing-token

new-line:
  the new-line character

defined-macro-expression:
  defined identifier
  defined ( identifier )

h-preprocessing-token:
  any preprocessing-token other than >

h-pp-tokens:
  h-preprocessing-token
  h-pp-tokens h-preprocessing-token

header-name-tokens:
  string-literal
  < h-pp-tokens >

has-include-expression:
  __has_include ( header-name )
  __has_include ( header-name-tokens )

has-attribute-expression:
  __has_cpp_attribute ( pp-tokens )

pp-module:
  export_opt module pp-tokens_opt ; new-line

pp-import:
  export_opt import header-name pp-tokens_opt ; new-line
  export_opt import header-name-tokens pp-tokens_opt ; new-line
  export_opt import pp-tokens ; new-line

va-opt-replacement:
  __VA_OPT__ ( pp-tokens_opt )
Annex B  (normative)
Implementation quantities  [implimits]

Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall document those limitations where known. This documentation may cite fixed limits where they exist, say how to compute variable limits as a function of available resources, or say that fixed limits do not exist or are unknown.

The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.

(2.1) — Nesting levels of compound statements (8.4), iteration control structures (8.6), and selection control structures (8.5) [256].
(2.2) — Nesting levels of conditional inclusion (15.2) [256].
(2.3) — Pointer (9.3.4.2), array (9.3.4.5), and function (9.3.4.6) declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration [256].
(2.4) — Nesting levels of parenthesized expressions (7.5.3) within a full-expression [256].
(2.5) — Number of characters in an internal identifier (5.10) or macro name (15.6) [1024].
(2.6) — Number of characters in an external identifier (5.10, 6.6) [1024].
(2.7) — External identifiers (6.6) in one translation unit [65536].
(2.8) — Identifiers with block scope declared in one block (6.4.3) [1024].
(2.9) — Structured bindings (9.6) introduced in one declaration [256].
(2.10) — Macro identifiers (15.6) simultaneously defined in one translation unit [65536].
(2.11) — Parameters in one function definition (9.5.1) [256].
(2.12) — Arguments in one function call (7.6.1.3) [256].
(2.13) — Parameters in one macro definition (15.6) [256].
(2.14) — Arguments in one macro invocation (15.6) [256].
(2.15) — Characters in one logical source line (5.2) [65536].
(2.16) — Characters in a string-literal (5.13.5) (after concatenation (5.2)) [65536].
(2.17) — Size of an object (6.7.2) [262144].
(2.18) — Nesting levels for #include files (15.3) [256].
(2.19) — Case labels for a switch statement (8.5.3) (excluding those for any nested switch statements) [16384].
(2.20) — Non-static data members (including inherited ones) in a single class (11.4) [16384].
(2.21) — Lambda-captures in one lambda-expression (7.5.5.3) [256].
(2.22) — Enumeration constants in a single enumeration (9.7.1) [4096].
(2.23) — Levels of nested class definitions (11.4.11) in a single member-specification [256].
(2.24) — Functions registered by atexit() (17.5) [32].
(2.25) — Functions registered by at_quick_exit() (17.5) [32].
(2.26) — Direct and indirect base classes (11.7) [16384].
(2.27) — Direct base classes for a single class (11.7) [1024].
(2.28) — Class members declared in a single member-specification (including member functions) (11.4) [4096].
(2.29) — Final overriding virtual functions in a class, accessible or not (11.7.3) [16384].
(2.30) — Direct and indirect virtual bases of a class (11.7.2) [1024].
- Static data members of a class (11.4.9.3) [1024].
- Friend declarations in a class (11.9.4) [4096].
- Access control declarations in a class (11.9.2) [4096].
- Member initializers in a constructor definition (11.10.3) [6144].
- `initializer-clauses` in one `braced-init-list` (9.4) [16384].
- Scope qualifications of one identifier (7.5.4.3) [256].
- Nested `linkage-specifications` (9.11) [1024].
- Recursive constexpr function invocations (9.2.6) [512].
- Full-expressions evaluated within a core constant expression (7.7) [1048576].
- Template parameters in a template declaration (13.2) [1024].
- Recursively nested template instantiations (13.9.2), including substitution during template argument deduction (13.10.3) [1024].
- Handlers per try block (14.4) [256].
- Number of placeholders (20.14.15.5) [10].
Annex C  (informative)
Compatibility

C.1  C++ and ISO C++ 2017

C.1.1  General
Subclause C.1 lists the differences between C++ and ISO C++ 2017 (ISO/IEC 14882:2017, Programming Languages — C++), by the chapters of this document.

C.1.2  Clause 5: lexical conventions
Affected subclauses: 5.4, 10.1, 10.3, 15.1, 15.4, and 15.5
Change: New identifiers with special meaning.
Rationale: Required for new features.
Effect on original feature: Logical lines beginning with module or import may be interpreted differently in this revision of C++.

Example 1:
```cpp
class module {}
module m1;  // was variable declaration; now module-declaration
module *m2;  // variable declaration

class import {}
import j1;  // was variable declaration; now module-import-declaration
::import j2;  // variable declaration
```
—end example

Example 2:
```cpp
template<typename> class import {}
import<int> f();  // ill-formed; previously well-formed
::import<int> g();  // OK
```
—end example

Affected subclause: 5.11
Change: New keywords.
Rationale: Required for new features.

(3.1) The char8_t keyword is added to differentiate the types of ordinary and UTF-8 literals (5.13.5).
(3.2) The concept keyword is added to enable the definition of concepts (13.7.9).
(3.3) The constexpr keyword is added to declare immediate functions (9.2.6).
(3.4) The constinit keyword is added to prevent unintended dynamic initialization (9.2.7).
(3.5) The co_await, co_yield, and co_return keywords are added to enable the definition of coroutines (9.5.4).
(3.6) The requires keyword is added to introduce constraints through a requires-clause (13.1) or a requires-expression (7.5.7).

Effect on original feature: Valid C++ 2017 code using char8_t, concept, constexpr, constinit, co_await, co_yield, co_return, or requires as an identifier is not valid in this revision of C++.

Affected subclause: 5.12
Change: New operator <=>.
Rationale: Necessary for new functionality.
**Effect on original feature:** Valid C++ 2017 code that contains a \(<=\) token immediately followed by a \(>\) token may be ill-formed or have different semantics in this revision of C++:

```c
namespace N {
    struct X {
        bool operator<=(X, X);
        template<bool(X, X)> struct Y {
            Y<operator<=> y;       // ill-formed; previously well-formed
        }
    }
}
```

**Affected subclause:** 5.13

**Change:** Type of UTF-8 string and character literals.

**Rationale:** Required for new features. The changed types enable function overloading, template specialization, and type deduction to distinguish ordinary and UTF-8 string and character literals.

**Effect on original feature:** Valid C++ 2017 code that depends on UTF-8 string literals having type “array of const char” and UTF-8 character literals having type “char” is not valid in this revision of C++.

```c
const auto *u8s = u8"text";       // u8s previously deduced as const char*; now deduced as const char8_t*
const char *ps = u8s;            // ill-formed; previously well-formed
auto u8c = u8%c;                 // u8c previously deduced as char; now deduced as char8_t
char *pc = &u8c;                 // ill-formed; previously well-formed
std::string s = u8"text";       // ill-formed; previously well-formed
void f(const char *s);
f(u8"text");                    // ill-formed; previously well-formed
```

### C.1.3 Clause 6: basics

**Affected subclause:** 6.7.3

**Change:** A pseudo-destructor call ends the lifetime of the object to which it is applied.

**Rationale:** Increase consistency of the language model.

**Effect on original feature:** Valid ISO C++ 2017 code may be ill-formed or have undefined behavior in this revision of C++.

```c
int f() {
    int a = 123;
    using T = int;
    a.-T();
    return a;          // undefined behavior; previously returned 123
}
```

— end example]

**Affected subclause:** 6.9.2.2

**Change:** Except for the initial release operation, a release sequence consists solely of atomic read-modify-write operations.

**Rationale:** Removal of rarely used and confusing feature.

**Effect on original feature:** If a \(\text{memory\_order\_release}\) atomic store is followed by a \(\text{memory\_order\_relaxed}\) store to the same variable by the same thread, then reading the latter value with a \(\text{memory\_order\_acquire}\) load no longer provides any “happens before” guarantees, even in the absence of intervening stores by another thread.

### C.1.4 Clause 7: expressions

**Affected subclause:** 7.5.5.3

**Change:** Implicit lambda capture may capture additional entities.

**Rationale:** Rule simplification, necessary to resolve interactions with constexpr if.
**Effect on original feature:** Lambdas with a *capture-default* may capture local entities that were not captured in C++ 2017 if those entities are only referenced in contexts that do not result in an odr-use.

### C.1.5 Clause 9: declarations

1. **Affected subclause:** 9.2.4
   **Change:** Unnamed classes with a typedef name for linkage purposes can contain only C-compatible constructs.
   **Rationale:** Necessary for implementability.
   **Effect on original feature:** Valid C++ 2017 code may be ill-formed in this revision of C++.
   ```
   typedef struct {
     void f() {} // ill-formed; previously well-formed
   } S;
   ```

2. **Affected subclause:** 9.3.4.7
   **Change:** A function cannot have different default arguments in different translation units.
   **Rationale:** Required for modules support.
   **Effect on original feature:** Valid C++ 2017 code may be ill-formed in this revision of C++, with no diagnostic required.
   ```
   // Translation unit 1
   int f(int a = 42);
   int g() { return f(); }

   // Translation unit 2
   int f(int a = 76) { return a; } // ill-formed, no diagnostic required; previously well-formed
   int g();
   int main() { return g(); } // used to return 42
   ```

3. **Affected subclause:** 9.4.2
   **Change:** A class that has user-declared constructors is never an aggregate.
   **Rationale:** Remove potentially error-prone aggregate initialization which may apply notwithstanding the declared constructors of a class.
   **Effect on original feature:** Valid C++ 2017 code that aggregate-initializes a type with a user-declared constructor may be ill-formed or have different semantics in this revision of C++.
   ```
   struct A {
     A() = delete; // not an aggregate; previously an aggregate
   };

   struct B {
     B() = default;
     int i = 0;
   };

   struct C {
     C(C&&) = default;
     int a, b;
   };

   A a{}; // ill-formed; previously well-formed
   B b = {1}; // ill-formed; previously well-formed
   auto* c = new C{2, 3}; // ill-formed; previously well-formed

   struct Y;

   struct X {
     operator Y();
   };

   struct Y { // not an aggregate; previously an aggregate
     Y(const Y&) = default;
     X x;
   };
   ```

§ C.1.5
Y y{x()};  // copy constructor call; previously aggregate-initialization

4 Affected subclause: 9.4.5
Change: Boolean conversion from a pointer or pointer-to-member type is now a narrowing conversion.
Rationale: Catches bugs.
Effect on original feature: Valid C++ 2017 code may fail to compile in this revision of C++. For example:

```cpp
bool y[1] = { "bc" };  // ill-formed; previously well-formed
```

C.1.6 Clause 11: classes

1 Affected subclauses: 11.4.5 and 11.4.8.3
Change: The class name can no longer be used parenthesized immediately after an `explicit decl-specifier` in a constructor declaration. The `conversion-function-id` can no longer be used parenthesized immediately after an `explicit decl-specifier` in a conversion function declaration.
Rationale: Necessary for new functionality.
Effect on original feature: Valid C++ 2017 code may fail to compile in this revision of C++. For example:

```cpp
struct S {
    explicit (S)(const S&);  // ill-formed; previously well-formed
    explicit (operator int)();  // ill-formed; previously well-formed
    explicit(true) (S)(int);  // OK
};
```

2 Affected subclauses: 11.4.5 and 11.4.7
Change: A `simple-template-id` is no longer valid as the `declarator-id` of a constructor or destructor.
Rationale: Remove potentially error-prone option for redundancy.
Effect on original feature: Valid C++ 2017 code may fail to compile in this revision of C++. For example:

```cpp
template<class T>
struct A {
    A<T>();  // error: simple-template-id not allowed for constructor
    A(int);  // OK, injected-class-name used
    ~A<T>();  // error: simple-template-id not allowed for destructor
};
```

3 Affected subclause: 11.10.6
Change: A function returning an implicitly movable entity may invoke a constructor taking an rvalue reference to a type different from that of the returned expression. Function and catch-clause parameters can be thrown using move constructors.
Rationale: Side effect of making it easier to write more efficient code that takes advantage of moves.
Effect on original feature: Valid C++ 2017 code may fail to compile or have different semantics in this revision of C++. For example:

```cpp
struct base {
    base();
    base(base const &);  
private:
    base(base &&);
};

struct derived : base {};

base f(base b) {
    throw b;  // error: base(base &&) is private
    derived d;
    return d;  // error: base(base &&) is private
}
```

```cpp
struct S {
    S(const char *s) : m(s) { }
    S(const S&) = default;
    S(S&& other) : m(other.m) { other.m = nullptr; }
        const char * m;
};

S consume(S&& s) { return s; }
```
void g() {
    S s("text");
    consume(static_cast<S&&>(s));
    char c = *s.m;  // undefined behavior; previously ok
}

C.1.7 Clause 12: overloading

Affected subclause: 12.4.2.3

Change: Equality and inequality expressions can now find reversed and rewritten candidates.

Rationale: Improve consistency of equality with three-way comparison and make it easier to write the full complement of equality operations.

Effect on original feature: Equality and inequality expressions between two objects of different types, where one is convertible to the other, could invoke a different operator. Equality and inequality expressions between two objects of the same type could become ambiguous.

struct A {
    operator int() const;
};

bool operator==(A, int);  // #1
// #2 is built-in candidate: bool operator==(int, int);
// #3 is built-in candidate: bool operator!=(int, int);

int check(A x, A y) {
    return (x == y) +  // ill-formed; previously well-formed
        (10 == x) +  // calls #1, previously selected #2
        (10 != x);  // calls #1, previously selected #3
}

C.1.8 Clause 13: templates

Affected subclause: 13.3

Change: An unqualified-id that is followed by a < and for which name lookup finds nothing or finds a function will be treated as a template-name in order to potentially cause argument dependent lookup to be performed.

Rationale: It was problematic to call a function template with an explicit template argument list via argument dependent lookup because of the need to have a template with the same name visible via normal lookup.

Effect on original feature: Previously valid code that uses a function name as the left operand of a < operator would become ill-formed.

struct A {};
bool operator<(void (*fp)(), A);
void f() {}
int main() {
    A a;
    f < a;  // ill-formed; previously well-formed
    (f) < a;  // still well formed
}

C.1.9 Clause 14: exception handling

Affected subclause: 14.5

Change: Remove throw() exception specification.

Rationale: Removal of obsolete feature that has been replaced by noexcept.

Effect on original feature: A valid C++ 2017 function declaration, member function declaration, function pointer declaration, or function reference declaration that uses throw() for its exception specification will be rejected as ill-formed in this revision of C++. It should simply be replaced with noexcept for no change of meaning since C++ 2017.

[Note 1: There is no way to write a function declaration that is non-throwing in this revision of C++ and is also non-throwing in C++ 2003 except by using the preprocessor to generate a different token sequence in each case. — end note]
C.1.10  Clause 16: library introduction

1. Affected subclause: 16.4.2.3
   Change: New headers.
   Rationale: New functionality.
   Effect on original feature: The following C++ headers are new: `<barrier>` (32.8.3.2), `<bit>` (26.5.2), `<charconv>` (20.19.1), `<compare>` (17.11.1), `<concepts>` (18.3), `<coroutine>` (17.12.2), `<format>` (20.20.1), `<latch>` (32.8.2.2), `<numbers>` (26.9.1), `<ranges>` (24.2), `<semaphore>` (32.7.2), `<source_location>` (17.8.1), `<span>` (22.7.2), `<stop_token>` (32.3.2), and `<version>` (17.3.1). Valid C++ 2017 code that `#include`s headers with these names may be invalid in this revision of C++.

2. Affected subclause: 16.4.2.3
   Change: Remove vacuous C++ header files.
   Rationale: The empty headers implied a false requirement to achieve C compatibility with the C++ headers.
   Effect on original feature: A valid C++ 2017 program that `#include`s any of the following headers may fail to compile: `<ccomplex>`, `<ciso646>`, `<cstdalign>`, `<cstdbool>`, and `<ctgmath>`. To retain the same behavior:

   1. a `#include` of `<ccomplex>` can be replaced by a `#include` of `<complex>` (26.4.2),
   2. a `#include` of `<ctgmath>` can be replaced by a `#include` of `<cmath>` (26.8.1) and a `#include` of `<complex>`, and
   3. a `#include` of `<ciso646>`, `<cstdalign>`, or `<cstdbool>` can simply be removed.

C.1.11  Clause 22: containers library

1. Affected subclauses: 22.3.9 and 22.3.10
   Change: Return types of `remove`, `remove_if`, and `unique` changed from `void` to `container::size_type`.
   Rationale: Improve efficiency and convenience of finding number of removed elements.
   Effect on original feature: Code that depends on the return types might have different semantics in this revision of C++. Translation units compiled against this version of C++ may be incompatible with translation units compiled against C++ 2017, either failing to link or having undefined behavior.

C.1.12  Clause 23: iterators library

1. Affected subclause: 23.3.2.3
   Change: The specialization of `iterator_traits` for `void*` and for function pointer types no longer contains any nested typedefs.
   Rationale: Corrects an issue misidentifying pointer types that are not incrementable as iterator types.
   Effect on original feature: A valid C++ 2017 program that relies on the presence of the typedefs may fail to compile, or have different behavior.

C.1.13  Clause 25: algorithms library

1. Affected subclause: 25.2
   Change: The number and order of deducible template parameters for algorithm declarations is now unspecified, instead of being as-declared.
   Rationale: Increase implementor freedom and allow some function templates to be implemented as function objects with templated call operators.
   Effect on original feature: A valid C++ 2017 program that passes explicit template arguments to algorithms not explicitly specified to allow such in this version of C++ may fail to compile or have undefined behavior.

C.1.14  Clause 29: input/output library

1. Affected subclause: 29.7.4.3.3
   Change: Character array extraction only takes array types.
   Rationale: Increase safety via preventing buffer overflow at compile time.
   Effect on original feature: Valid C++ 2017 code may fail to compile in this revision of C++:

   ```cpp
   auto p = new char[100];
   char q[100];
   std::cin >> std::setw(20) >> p; // ill-formed; previously well-formed
   std::cin >> std::setw(20) >> q; // OK
   ```

2. Affected subclause: 29.7.5.3.4
   Change: Overload resolution for ostream inserters used with UTF-8 literals.
Rationale: Required for new features.

Effect on original feature: Valid C++ 2017 code that passes UTF-8 literals to `basic_ostream<char, ...>::operator<<` or `basic_ostream<wchar_t, ...>::operator<<` is now ill-formed.

```cpp
std::cout << u8"text"; // previously called operator<<(const char*) and printed a string; now ill-formed
std::cout << u8'X'; // previously called operator<<(char) and printed a character; now ill-formed
```

3 Affected subclause: 29.7.5.3.4

Change: Overload resolution for ostream inserters used with wchar_t, char16_t, or char32_t types.

Rationale: Removal of surprising behavior.

Effect on original feature: Valid C++ 2017 code that passes wchar_t, char16_t, or char32_t characters or strings to `basic_ostream<char, ...>::operator<<` or that passes char16_t or char32_t characters or strings to `basic_ostream<wchar_t, ...>::operator<<` is now ill-formed.

```cpp
std::cout << u8"text"; // previously formatted the string as a pointer value; now ill-formed
std::cout << u8'X'; // previously formatted the character as an integer value; now ill-formed
```

4 Affected subclause: 29.11.6

Change: Return type of filesystem path format observer member functions.

Rationale: Required for new features.

Effect on original feature: Valid C++ 2017 code that depends on the `u8string()` and `generic_u8string()` member functions of `std::filesystem::path` returning `std::string` is not valid in this revision of C++.

```cpp
std::filesystem::path p;
std::string s1 = p.u8string(); // ill-formed; previously well-formed
std::string s2 = p.generic_u8string(); // ill-formed; previously well-formed
```

C.1.15 Annex D: compatibility features

[diff.cpp17.depr]

1 Change: Remove `uncaught_exception`.

Rationale: The function did not have a clear specification when multiple exceptions were active, and has been superseded by `uncaught_exceptions`.

Effect on original feature: A valid C++ 2017 program that calls `std::uncaught_exception` may fail to compile. It might be revised to use `std::uncaught_exceptions` instead, for clear and portable semantics.

2 Change: Remove support for adaptable function API.

Rationale: The deprecated support relied on a limited convention that could not be extended to support the general case or new language features. It has been superseded by direct language support with `decltype`, and by the `std::bind` and `std::not_fn` function templates.

Effect on original feature: A valid C++ 2017 program that relies on the presence of `result_type`, `argument_type`, `first_argument_type`, or `second_argument_type` in a standard library class may fail to compile. A valid C++ 2017 program that calls `not1` or `not2`, or uses the class templates `unary_negate` or `binary_negate`, may fail to compile.

3 Change: Remove redundant members from `std::allocator`.

Rationale: `std::allocator` was overspecified, encouraging direct usage in user containers rather than relying on `std::allocator_traits`, leading to poor containers.

Effect on original feature: A valid C++ 2017 program that directly makes use of the `pointer`, `const_pointer`, `reference`, `const_reference`, `rebind`, `address`, `construct`, `destroy`, or `max_size` members of `std::allocator`, or that directly calls `allocate` with an additional hint argument, may fail to compile.

4 Change: Remove `raw_storage_iterator`.

Rationale: The iterator encouraged use of algorithms that might throw exceptions, but did not return the number of elements successfully constructed that might need to be destroyed in order to avoid leaks.

Effect on original feature: A valid C++ 2017 program that uses this iterator class may fail to compile.

5 Change: Remove temporary buffers API.

Rationale: The temporary buffer facility was intended to provide an efficient optimization for small memory requests, but there is little evidence this was achieved in practice, while requiring the user to provide their own exception-safe wrappers to guard use of the facility in many cases.
Effect on original feature: A valid C++ 2017 program that calls get_temporary_buffer or return_-
temporary_buffer may fail to compile.

Change: Remove shared_ptr::unique.
Rationale: The result of a call to this member function is not reliable in the presence of multiple threads
and weak pointers. The member function use_count is similarly unreliable, but has a clearer contract in
such cases, and remains available for well defined use in single-threaded cases.
Effect on original feature: A valid C++ 2017 program that calls unique on a shared_ptr object may fail
to compile.

Affected subclause: D.14
Change: Remove deprecated type traits.
Rationale: The traits had unreliable or awkward interfaces. The is_literal_type trait provided no way
to detect which subset of constructors and member functions of a type were declared constexpr. The
result_of trait had a surprising syntax that could not report the result of a regular function type. It has
been superseded by the invoke_result trait.
Effect on original feature: A valid C++ 2017 program that relies on the is_literal_type or result_of
type traits, on the is_literal_type_v variable template, or on the result_of_t alias template may fail to
compile.

C.2 C++ and ISO C++ 2014

C.2.1 General

Subclause C.2 lists the differences between C++ and ISO C++ 2014 (ISO/IEC 14882:2014, Programming
Languages — C++), in addition to those listed above, by the chapters of this document.

C.2.2 Clause 5: lexical conventions

Affected subclause: 5.2
Change: Removal of trigraph support as a required feature.
Rationale: Prevents accidental uses of trigraphs in non-raw string literals and comments.
Effect on original feature: Valid C++ 2014 code that uses trigraphs may not be valid or may have different
semantics in this revision of C++. Implementations may choose to translate trigraphs as specified in C++ 2014
if they appear outside of a raw string literal, as part of the implementation-defined mapping from physical
source file characters to the basic source character set.

Affected subclause: 5.9
Change: pp-number can contain p sign and P sign.
Rationale: Necessary to enable hexadecimal-floating-point-literals.
Effect on original feature: Valid C++ 2014 code may fail to compile or produce different results in this
revision of C++. Specifically, character sequences like 0p+0 and 0e1_p+0 are three separate tokens each in
C++ 2014, but one single token in this revision of C++. For example:

```cpp
#define F(a) b ## a
int b0p = F(0p+0);  // ill-formed; equivalent to “int b0p = b0p + 0;” in C++ 2014
```

C.2.3 Clause 7: expressions

Affected subclauses: 7.6.1.6 and 7.6.2.3
Change: Remove increment operator with bool operand.
Rationale: Obsolete feature with occasionally surprising semantics.
Effect on original feature: A valid C++ 2014 expression utilizing the increment operator on a bool lvalue
is ill-formed in this revision of C++. Note that this might occur when the lvalue has a type given by a
template parameter.

Affected subclauses: 7.6.2.8 and 7.6.2.9
Change: Dynamic allocation mechanism for over-aligned types.
Rationale: Simplify use of over-aligned types.
Effect on original feature: In C++ 2014 code that uses a new-expression to allocate an object with an over-
aligned class type, where that class has no allocation functions of its own, ::operator new(std::size_t) is
used to allocate the memory. In this revision of C++, ::operator new(std::size_t, std::align_val_t) is
used instead.
C.2.4 Clause 9: declarations

1 Affected subclause: 9.2.2
   Change: Removal of register storage-class-specifier.
   Rationale: Enable repurposing of deprecated keyword in future revisions of C++.
   Effect on original feature: A valid C++ 2014 declaration utilizing the register storage-class-specifier is ill-formed in this revision of C++. The specifier can simply be removed to retain the original meaning.

2 Affected subclause: 9.2.9.6
   Change: auto deduction from braced-init-list.
   Rationale: More intuitive deduction behavior.
   Effect on original feature: Valid C++ 2014 code may fail to compile or may change meaning in this revision of C++. For example:
   
   ```cpp
   auto x1{1}; // was std::initializer_list<int>, now int
   auto x2{1, 2}; // was std::initializer_list<int>, now ill-formed
   ```

3 Affected subclause: 9.3.4.6
   Change: Make exception specifications be part of the type system.
   Rationale: Improve type-safety.
   Effect on original feature: Valid C++ 2014 code may fail to compile or change meaning in this revision of C++. For example:
   
   ```cpp
   void g1() noexcept;
   void g2();
   template<class T> int f(T *, T *);
   int x = f(g1, g2); // illexformed; previously well-formed
   ```

4 Affected subclause: 9.4.2
   Change: Definition of an aggregate is extended to apply to user-defined types with base classes.
   Rationale: To increase convenience of aggregate initialization.
   Effect on original feature: Valid C++ 2014 code may fail to compile or produce different results in this revision of C++; initialization from an empty initializer list will perform aggregate initialization instead of invoking a default constructor for the affected types. For example:
   
   ```cpp
   struct derived;
   struct base {
      friend struct derived;
   private:
      base();
   };
   struct derived : base {};
   
   derived d1{}; // error; the code was well-formed in C++ 2014
   derived d2; // still OK
   ```

C.2.5 Clause 11: classes

1 Affected subclause: 11.10.4
   Change: Inheriting a constructor no longer injects a constructor into the derived class.
   Rationale: Better interaction with other language features.
   Effect on original feature: Valid C++ 2014 code that uses inheriting constructors may not be valid or may have different semantics. A using-declaration that names a constructor now makes the corresponding base class constructors visible to initializations of the derived class rather than declaring additional derived class constructors.
   
   ```cpp
   struct A {
      template<typename T> A(T, typename T::type = 0);
      A(int);
   };
   struct B : A {
      using A::A;
      B(int);
   };
   B b(42L); // now calls B(int), used to call B<long>(long),
   // which called A<int> due to substitution failure
   // in A<long>(long).
   ```
C.2.6 Clause 13: templates

Affected subclause: 13.10.3.6
Change: Allowance to deduce from the type of a non-type template argument.
Rationale: In combination with the ability to declare non-type template arguments with placeholder types, allows partial specializations to decompose from the type deduced for the non-type template argument.
Effect on original feature: Valid C++ 2014 code may fail to compile or produce different results in this revision of C++. For example:

```cpp
template <int N> struct A;
template <typename T, T N> int foo(A<N> *) = delete;
void foo(void *);
void bar(A<0> *p) {
    foo(p);
    // ill-formed; previously well-formed
}
```

C.2.7 Clause 14: exception handling

Affected subclause: 14.5
Change: Remove dynamic exception specifications.
Rationale: Dynamic exception specifications were a deprecated feature that was complex and brittle in use. They interacted badly with the type system, which became a more significant issue in this revision of C++ where (non-dynamic) exception specifications are part of the function type.
Effect on original feature: A valid C++ 2014 function declaration, member function declaration, function pointer declaration, or function reference declaration, if it has a potentially throwing dynamic exception specification, is rejected as ill-formed in this revision of C++. Violating a non-throwing dynamic exception specification calls `terminate` rather than `unexpected` and might not perform stack unwinding prior to such a call.

C.2.8 Clause 16: library introduction

Affected subclause: 16.4.2.3
Change: New headers.
Rationale: New functionality.
Effect on original feature: The following C++ headers are new: `<any>` (20.8.2), `<charconv>` (20.19.1), `<execution>` (20.18.2), `<filesystem>` (29.11.4), `<memory_resource>` (20.12.1), `<optional>` (20.6.2), `<string_view>` (21.4.2), and `<variant>` (20.7.2). Valid C++ 2014 code that `#include`s headers with these names may be invalid in this revision of C++.

Affected subclause: 16.4.5.2.3
Change: New reserved namespaces.
Rationale: Reserve namespaces for future revisions of the standard library that might otherwise be incompatible with existing programs.
Effect on original feature: The global namespaces `std` followed by an arbitrary sequence of `digits` (5.10) are reserved for future standardization. Valid C++ 2014 code that uses such a top-level namespace, e.g., `std2`, may be invalid in this revision of C++.

C.2.9 Clause 20: general utilities library

Affected subclause: 20.14.17
Change: Constructors taking allocators removed.
Rationale: No implementation consensus.
Effect on original feature: Valid C++ 2014 code may fail to compile or may change meaning in this revision of C++. Specifically, constructing a `std::function` with an allocator is ill-formed and uses-allocator construction will not pass an allocator to `std::function` constructors in this revision of C++.

Affected subclause: 20.11.3
Change: Different constraint on conversions from `unique_ptr`.
Rationale: Adding array support to `shared_ptr`, via the syntax `shared_ptr<T[]>` and `shared_ptr<T[N]>`.
Effect on original feature: Valid C++ 2014 code may fail to compile or may change meaning in this revision of C++. For example:

```cpp
#include <memory>
std::unique_ptr<int[]> arr(new int[1]);
std::shared_ptr<int> ptr(std::move(arr)); // error: int(*)[] is not compatible with int*
```
C.2.10 Clause 21: strings library

Affected subclause: 21.3.3
Change: Non-const .data() member added.
Rationale: The lack of a non-const .data() differed from the similar member of std::vector. This change regularizes behavior.
Effect on original feature: Overloaded functions which have differing code paths for char* and const char* arguments will execute differently when called with a non-const string's .data() member in this revision of C++.

```cpp
int f(char *) = delete;
int f(const char *);
string s;
int x = f(s.data()); // ill-formed; previously well-formed
```

C.2.11 Clause 22: containers library

Affected subclause: 22.2.6
Change: Requirements change:
Rationale: Increase portability, clarification of associative container requirements.
Effect on original feature: Valid C++ 2014 code that attempts to use associative containers having a comparison object with non-const function call operator may fail to compile in this revision of C++:

```cpp
#include <set>
struct compare{
    bool operator()(int a, int b)
    {
        return a < b;
    }
};
int main() {
    const std::set<int, compare> s;
    s.find(0);
}
```

C.2.12 Annex D: compatibility features

Change: The class templates auto_ptr, unary_function, and binary_function, the function templates random_shuffle, and the function templates (and their return types) ptr_fun, mem_fun, mem_fun_ref, bind1st, and bind2nd are not defined.
Rationale: Superseded by new features.
Effect on original feature: Valid C++ 2014 code that uses these class templates and function templates may fail to compile in this revision of C++.

Change: Remove old iostreams members [depr.ios.members].
Rationale: Redundant feature for compatibility with pre-standard code has served its time.
Effect on original feature: A valid C++ 2014 program using these identifiers may be ill-formed in this revision of C++.

C.3 C++ and ISO C++ 2011

C.3.1 General

Subclause C.3 lists the differences between C++ and ISO C++ 2011 (ISO/IEC 14882:2011, Programming Languages — C++), in addition to those listed above, by the chapters of this document.

C.3.2 Clause 5: lexical conventions
N4868

© ISO/IEC

the macro invocation produces different outcomes because the single quotes delimit a character-literal in C++
2011, whereas they are digit separators in this revision of C++:
#define M(x, ...) __VA_ARGS__
int x[2] = { M(1’2,3’4, 5) };
// int x[2] = { 5 };
— C++ 2011
// int x[2] = { 3’4, 5 }; — this revision of C++

C.3.3
1

Clause 6: basics

[diff.cpp11.basic]

Affected subclause: 6.7.5.5.3
Change: New usual (non-placement) deallocator.
Rationale: Required for sized deallocation.
Effect on original feature: Valid C++ 2011 code could declare a global placement allocation function and
deallocation function as follows:
void* operator new(std::size_t, std::size_t);
void operator delete(void*, std::size_t) noexcept;

In this revision of C++, however, the declaration of operator delete might match a predefined usual
(non-placement) operator delete (6.7.5.5). If so, the program is ill-formed, as it was for class member
allocation functions and deallocation functions (7.6.2.8).

C.3.4
1

Clause 7: expressions

[diff.cpp11.expr]

Affected subclause: 7.6.16
Change: A conditional expression with a throw expression as its second or third operand keeps the type
and value category of the other operand.
Rationale: Formerly mandated conversions (lvalue-to-rvalue (7.3.2), array-to-pointer (7.3.3), and functionto-pointer (7.3.4) standard conversions), especially the creation of the temporary due to lvalue-to-rvalue
conversion, were considered gratuitous and surprising.
Effect on original feature: Valid C++ 2011 code that relies on the conversions may behave differently in
this revision of C++:
struct S {
int x = 1;
void mf() { x = 2; }
};
int f(bool cond) {
S s;
(cond ? s : throw 0).mf();
return s.x;
}

In C++ 2011, f(true) returns 1. In this revision of C++, it returns 2.
sizeof(true ? "" : throw 0)

In C++ 2011, the expression yields sizeof(const char*). In this revision of C++, it yields sizeof(const
char[1]).

C.3.5
1

Clause 9: declarations

[diff.cpp11.dcl.dcl]

Affected subclause: 9.2.6
Change: constexpr non-static member functions are not implicitly const member functions.
Rationale: Necessary to allow constexpr member functions to mutate the object.
Effect on original feature: Valid C++ 2011 code may fail to compile in this revision of C++. For example,
the following code is valid in C++ 2011 but invalid in this revision of C++ because it declares the same
member function twice with different return types:
struct S {
constexpr const int &f();
int &f();
};

2

Affected subclause: 9.4.2
Change: Classes with default member initializers can be aggregates.
Rationale: Necessary to allow default member initializers to be used by aggregate initialization.

§ C.3.5

1664


Effect on original feature: Valid C++ 2011 code may fail to compile or may change meaning in this revision of C++. For example:

```cpp
struct S {
    int m = 1;
};
struct X {
    operator int();
    operator S();
};
X a{};
S b(a);
// uses copy constructor in C++ 2011,
// performs aggregate initialization in this revision of C++
```

C.3.6 Clause 16: library introduction

1 Affected subclause: 16.4.2.3
   Change: New header.
   Rationale: New functionality.
   Effect on original feature: The C++ header `<shared_mutex>` (32.5.3) is new. Valid C++ 2011 code that
   #includes with that name may be invalid in this revision of C++.

C.3.7 Clause 29: input/output library

1 Affected subclause: 29.12
   Change: `gets` is not defined.
   Rationale: Use of `gets` is considered dangerous.
   Effect on original feature: Valid C++ 2011 code that uses the `gets` function may fail to compile in this
   revision of C++.

C.4 C++ and ISO C++ 2003

C.4.1 General

1 Subclause C.4 lists the differences between C++ and ISO C++ 2003 (ISO/IEC 14882:2003, Programming
   Languages — C++), in addition to those listed above, by the chapters of this document.

C.4.2 Clause 5: lexical conventions

1 Affected subclause: 5.4
   Change: New kinds of string-literals.
   Rationale: Required for new features.
   Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this
   revision of C++. Specifically, macros named R, u8, u8R, u, uR, U, UR, or LR will not be expanded when adjacent
   to a string-literal but will be interpreted as part of the string-literal. For example:

   ```cpp
   #define u8 "abc"
   const char* s = u8"def"; // Previously "abc" "def"
   ``

2 Affected subclause: 5.4
   Change: User-defined literal string support.
   Rationale: Required for new features.
   Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this
   revision of C++. For example:

   ```cpp
   #define _x "there"
   "hello"_x // #1
   ```

Previously, #1 would have consisted of two separate preprocessing tokens and the macro _x would have been
expanded. In this revision of C++, #1 consists of a single preprocessing token, so the macro is not expanded.

3 Affected subclause: 5.11
   Change: New keywords.
   Rationale: Required for new features.
   Effect on original feature: Added to Table 5, the following identifiers are new keywords: `alignas`, `alignof`,
   char16_t, char32_t, constexpr, decInt, noexcept, nullptr, static_assert, and `thread_local`. Valid
   C++ 2003 code using these identifiers is invalid in this revision of C++.
Affected subclause: 5.13.2
Change: Type of integer literals.
Rationale: C99 compatibility.
Effect on original feature: Certain integer literals larger than can be represented by `long` could change from an unsigned integer type to `signed long long`.

C.4.3 Clause 7: expressions

Affected subclause: 7.3.12
Change: Only literals are integer null pointer constants.
Rationale: Removing surprising interactions with templates and constant expressions.
Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this revision of C++. For example:

```cpp
void f(void *);  // #1
void f(...);  // #2
template<int N> void g() {
    f(0*N);  // calls #2; used to call #1
}
```

Affected subclause: 7.6.5
Change: Specify rounding for results of integer `/` and `%`.
Rationale: Increase portability, C99 compatibility.
Effect on original feature: Valid C++ 2003 code that uses integer division rounds the result toward 0 or toward negative infinity, whereas this revision of C++ always rounds the result toward 0.

Affected subclause: 7.6.14
Change: `&&` is valid in a type-name.
Rationale: Required for new features.
Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this revision of C++. For example:

```cpp
bool b1 = new int && false;  // previously false, now ill-formed
struct S { operator int(); };  
bool b2 = &S::operator int && false;  // previously false, now ill-formed
```

C.4.4 Clause 9: declarations

Affected subclause: 9.2
Change: Remove `auto` as a storage class specifier.
Rationale: New feature.
Effect on original feature: Valid C++ 2003 code that uses the keyword `auto` as a storage class specifier may be invalid in this revision of C++. In this revision of C++, `auto` indicates that the type of a variable is to be deduced from its initializer expression.

Affected subclause: 9.4.5
Change: Narrowing restrictions in aggregate initializers.
Rationale: Catches bugs.
Effect on original feature: Valid C++ 2003 code may fail to compile in this revision of C++. For example, the following code is valid in C++ 2003 but invalid in this revision of C++ because `double` to `int` is a narrowing conversion:

```cpp
int x[] = { 2.0 };  
```

C.4.5 Clause 11: classes

Affected subclauses: 11.4.5.2, 11.4.7, 11.4.5.3, and 11.4.6
Change: Implicitly-declared special member functions are defined as deleted when the implicit definition would have been ill-formed.
Rationale: Improves template argument deduction failure.
Effect on original feature: A valid C++ 2003 program that uses one of these special member functions in a context where the definition is not required (e.g., in an expression that is not potentially evaluated) becomes ill-formed.

Affected subclause: 11.4.7
Change: User-declared destructors have an implicit exception specification.
Rationale: Clarification of destructor requirements.

Effect on original feature: Valid C++ 2003 code may execute differently in this revision of C++. In particular, destructors that throw exceptions will call `std::terminate` (without calling `std::unexpected`) if their exception specification is non-throwing.

C.4.6 Clause 13: templates

1. Affected subclause: 13.2
   Change: Remove `export`.
   Rationale: No implementation consensus.
   Effect on original feature: A valid C++ 2003 declaration containing `export` is ill-formed in this revision of C++.

2. Affected subclause: 13.4
   Change: Remove whitespace requirement for nested closing template right angle brackets.
   Rationale: Considered a persistent but minor annoyance. Template aliases representing non-class types would exacerbate whitespace issues.
   Effect on original feature: Change to semantics of well-defined expression. A valid C++ 2003 expression containing a right angle bracket (`""`) followed immediately by another right angle bracket may now be treated as closing two templates. For example, the following code is valid in C++ 2003 because `""` is a right-shift operator, but invalid in this revision of C++ because `""` closes two templates.

   ```cpp
   template <class T> struct X { }
   template <int N> struct Y { }
   X< Y< 1 >> 2 > > x;
   ```

3. Affected subclause: 13.8.5.2
   Change: Allow dependent calls of functions with internal linkage.
   Rationale: Overly constrained, simplify overload resolution rules.
   Effect on original feature: A valid C++ 2003 program could get a different result than in this revision of C++.

C.4.7 Clause 16: library introduction

1. Affected: Clause 16 – Clause 32
   Change: New reserved identifiers.
   Rationale: Required by new features.
   Effect on original feature: Valid C++ 2003 code that uses any identifiers added to the C++ standard library by later revisions of C++ may fail to compile or produce different results in this revision of C++.
   A comprehensive list of identifiers used by the C++ standard library can be found in the Index of Library Names in this document.

2. Affected subclause: 16.4.2.3
   Change: New headers.
   Rationale: New functionality.
   Effect on original feature: The following C++ headers are new: `<array>` (22.3.2), `<atomic>` (31.2), `<chrono>` (27.2), `<codecvt>` (D.21.2), `<condition_variable>` (32.6.2), `<forward_list>` (22.3.4), `<future>` (32.9.2), `<initializer_list>` (17.10.2), `<mutex>` (32.5.2), `<random>` (26.6.2), `<ratio>` (20.16.2), `<regex>` (30.3), `<scoped_allocator>` (20.13.1), `<system_error>` (19.5.2), `<thread>` (32.4.2), `<tuple>` (20.5.2), `<typeindex>` (20.17.1), `<type_traits>` (20.15.3), `<unordered_map>` (22.5.2), and `<unordered_set>` (22.5.3).
   In addition the following C compatibility headers are new: `<cfenv>` (26.3.1), `<cinttypes>` (29.12.2), `<cstdint>` (17.4.2), and `<cuchar>` (21.5.5). Valid C++ 2003 code that includes headers with these names may be invalid in this revision of C++.

3. Affected subclause: 16.4.4.3
   Effect on original feature: Function `swap` moved to a different header
   Rationale: Remove dependency on `<algorithm>` (25.4) for `swap`.
   Effect on original feature: Valid C++ 2003 code that has been compiled expecting `swap` to be in `<algorithm>` (25.4) may have to instead include `<utility>` (20.2.1).

4. Affected subclause: 16.4.5.2.2
   Change: New reserved namespace.
   Rationale: New functionality.
Effect on original feature: The global namespace `posix` is now reserved for standardization. Valid C++ 2003 code that uses a top-level namespace `posix` may be invalid in this revision of C++.

Affected subclause: 16.4.6.3
Change: Additional restrictions on macro names.
Rationale: Avoid hard to diagnose or non-portable constructs.
Effect on original feature: Names of attribute identifiers may not be used as macro names. Valid C++ 2003 code that defines `override`, `final`, `carries_dependency`, or `noreturn` as macros is invalid in this revision of C++.

C.4.8 Clause 17: language support library [diff.cpp03.language.support]

Affected subclause: 17.6.3.2
Change: `operator new` may throw exceptions other than `std::bad_alloc`.
Rationale: Consistent application of `noexcept`.
Effect on original feature: Valid C++ 2003 code that assumes that global `operator new` only throws `std::bad_alloc` may execute differently in this revision of C++. Valid C++ 2003 code that replaces the global replaceable `operator new` is ill-formed in this revision of C++, because the exception specification of `throw(std::bad_alloc)` was removed.

C.4.9 Clause 19: diagnostics library [diff.cpp03.diagnostics]

Affected subclause: 19.4
Change: Thread-local error numbers.
Rationale: Support for new thread facilities.
Effect on original feature: Valid but implementation-specific C++ 2003 code that relies on `errno` being the same across threads may change behavior in this revision of C++.

C.4.10 Clause 20: general utilities library [diff.cpp03.utilities]

Affected subclause: 20.10.5
Change: Minimal support for garbage-collected regions.
Rationale: Required by new feature.
Effect on original feature: Valid C++ 2003 code, compiled without traceable pointer support, that interacts with newer C++ code using regions declared reachable may have different runtime behavior.

Change: Standard function object types no longer derived from `std::unary_function` or `std::binary_function`.
Rationale: Superseded by new feature; `unary_function` and `binary_function` are no longer defined.
Effect on original feature: Valid C++ 2003 code that depends on function object types being derived from `unary_function` or `binary_function` may fail to compile in this revision of C++.

C.4.11 Clause 21: strings library [diff.cpp03.strings]

Affected subclause: 21.3
Change: `basic_string` requirements no longer allow reference-counted strings.
Rationale: Invalidation is subtly different with reference-counted strings. This change regularizes behavior.
Effect on original feature: Valid C++ 2003 code may execute differently in this revision of C++.

Affected subclause: 21.3.3.2
Change: Loosen `basic_string` invalidation rules.
Rationale: Allow small-string optimization.
Effect on original feature: Valid C++ 2003 code may execute differently in this revision of C++. Some `const` member functions, such as `data` and `c_str`, no longer invalidate iterators.

C.4.12 Clause 22: containers library [diff.cpp03.containers]

Affected subclause: 22.2
Change: Complexity of `size()` member functions now constant.
Rationale: Lack of specification of complexity of `size()` resulted in divergent implementations with inconsistent performance characteristics.
Effect on original feature: Some container implementations that conform to C++ 2003 may not conform to the specified `size()` requirements in this revision of C++. Adjusting containers such as `std::list` to the stricter requirements may require incompatible changes.
Affected subclause: 22.2
Change: Requirements change: relaxation.
Rationale: Clarification.
Effect on original feature: Valid C++ 2003 code that attempts to meet the specified container requirements may now be over-specified. Code that attempted to be portable across containers may need to be adjusted as follows:

1. not all containers provide `size()`; use `empty()` instead of `size() == 0`;
2. not all containers are empty after construction (`array`);
3. not all containers have constant complexity for `swap()` (`array`).

Affected subclause: 22.2
Change: Requirements change: default constructible.
Rationale: Clarification of container requirements.
Effect on original feature: Valid C++ 2003 code that attempts to explicitly instantiate a container using a user-defined type with no default constructor may fail to compile.

Affected subclauses: 22.2.3 and 22.2.6
Change: Signature changes: from `void` return types.
Rationale: Old signature threw away useful information that may be expensive to recalculate.
Effect on original feature: The following member functions have changed:

1. `erase(iter)` for `set`, `multiset`, `map`, `multimap`
2. `erase(begin, end)` for `set`, `multiset`, `map`, `multimap`
3. `insert(pos, num, val)` for `vector`, `deque`, `list`, `forward_list`
4. `insert(pos, beg, end)` for `vector`, `deque`, `list`, `forward_list`

Valid C++ 2003 code that relies on these functions returning `void` (e.g., code that creates a pointer to member function that points to one of these functions) will fail to compile with this revision of C++.

Affected subclauses: 22.2.3 and 22.2.6
Change: Signature changes: from `iterator` to `const_iterator` parameters.
Rationale: Overspecification.
Effect on original feature: The signatures of the following member functions changed from taking an `iterator` to taking a `const_iterator`:

1. `insert(iter, val)` for `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap`
2. `insert(pos, beg, end)` for `vector`, `deque`, `list`, `forward_list`
3. `erase(begin, end)` for `set`, `multiset`, `map`, `multimap`
4. all forms of `list::splice`
5. all forms of `list::merge`

Valid C++ 2003 code that uses these functions may fail to compile with this revision of C++.

Affected subclauses: 22.2.3 and 22.2.6
Change: Signature changes: `resize`.
Rationale: Performance, compatibility with move semantics.
Effect on original feature: For `vector`, `deque`, and `list` the fill value passed to `resize` is now passed by reference instead of by value, and an additional overload of `resize` has been added. Valid C++ 2003 code that uses this function may fail to compile with this revision of C++.

C.4.13 Clause 25: algorithms library

Effect on original feature: A valid C++ 2003 program may detect that an object with a valid but unspecified state has a different valid but unspecified state with this revision of C++. For example, `std::remove` and `std::remove_if` may leave the tail of the input sequence with a different set of values than previously.
C.4.14 Clause 26: numerics library

Affected subclause: 26.4
Change: Specified representation of complex numbers.
Rationale: Compatibility with C99.
Effect on original feature: Valid C++ 2003 code that uses implementation-specific knowledge about the binary representation of the required template specializations of std::complex may not be compatible with this revision of C++.

C.4.15 Clause 29: input/output library

Affected subclauses: 29.7.4.2.4, 29.7.5.2.4, and 29.5.5.4
Change: Specify use of explicit in existing boolean conversion functions.
Rationale: Clarify intentions, avoid workarounds.
Effect on original feature: Valid C++ 2003 code that relies on implicit boolean conversions will fail to compile with this revision of C++. Such conversions occur in the following conditions:

1. passing a value to a function that takes an argument of type bool;
2. using operator== to compare to false or true;
3. returning a value from a function with a return type of bool;
4. initializing members of type bool via aggregate initialization;
5. initializing a const bool& which would bind to a temporary object.

Affected subclause: 29.5.3.2.1
Change: Change base class of std::ios_base::failure.
Rationale: More detailed error messages.
Effect on original feature: std::ios_base::failure is no longer derived directly from std::exception, but is now derived from std::system_error, which in turn is derived from std::runtime_error. Valid C++ 2003 code that assumes that std::ios_base::failure is derived directly from std::exception may execute differently in this revision of C++.

Affected subclause: 29.5.3
Change: Flag types in std::ios_base are now bitmasks with values defined as constexpr static members.
Rationale: Required for new features.
Effect on original feature: Valid C++ 2003 code that relies on std::ios_base flag types being represented as std::bitset or as an integer type may fail to compile with this revision of C++. For example:

```cpp
#include <iostream>

int main() {
    int flag = std::ios_base::hex;
    std::cout.setf(flag); // error: setf does not take argument of type int
}
```

C.5 C++ and ISO C

C.5.1 General

Subclause C.5 lists the differences between C++ and ISO C, in addition to those listed above, by the chapters of this document.

C.5.2 Clause 5: lexical conventions

Affected subclause: 5.11
Change: New Keywords
New keywords are added to C++; see 5.11.
Rationale: These keywords were added in order to implement the new semantics of C++.
Effect on original feature: Change to semantics of well-defined feature. Any ISO C programs that used any of these keywords as identifiers are not valid C++ programs.
Difficulty of converting: Syntactic transformation. Converting one specific program is easy. Converting a large collection of related programs takes more work.
How widely used: Common.
Affected subclause: 5.13.3  
Change: Type of character-literal is changed from int to char.  
Rationale: This is needed for improved overloaded function argument type matching. For example:

```c
int function( int i );
int function( char c );
```

```c
function( 'x' );
```

It is preferable that this call match the second version of function rather than the first.  
Effect on original feature: Change to semantics of well-defined feature. ISO C programs which depend on

```c
sizeof('x') == sizeof(int)
```

will not work the same as C++ programs.  
Difficulty of converting: Simple.  
How widely used: Programs which depend upon `sizeof('x')` are probably rare.

Affected subclause: 5.13.5  
Change: String literals made const.  
The type of a string-literal is changed from “array of char” to “array of const char”. The type of a UTF-8 string literal is changed from “array of char” to “array of const char8_t”. The type of a UTF-16 string literal is changed from “array of some-integer-type” to “array of const char16_t”. The type of a UTF-32 string literal is changed from “array of some-integer-type” to “array of const char32_t”. The type of a wide string literal is changed from “array of wchar_t” to “array of const wchar_t”.  
Rationale: This avoids calling an inappropriate overloaded function, which might expect to be able to modify its argument.  
Effect on original feature: Change to semantics of well-defined feature.  
Difficulty of converting: Syntactic transformation. The fix is to add a cast:

```c
char* p = "abc";  // valid in C, invalid in C++
void f(char*) {  
    char* p = (char*)"abc";  // OK: cast added
    f(p);
    f((char*)"def");  // OK: cast added
}
```

How widely used: Programs that have a legitimate reason to treat string literal objects as potentially modifiable memory are probably rare.

C.5.3 Clause 6: basics  

Affected subclause: 6.2  
Change: C++ does not have “tentative definitions” as in C. E.g., at file scope,  

```c
int i;
int i;
```

is valid in C, invalid in C++. This makes it impossible to define mutually referential file-local objects with static storage duration, if initializers are restricted to the syntactic forms of C. For example,

```c
struct X { int i; struct X* next; };

static struct X a;
static struct X b = { 0, &a };
static struct X a = { 1, &b };
```

Rationale: This avoids having different initialization rules for fundamental types and user-defined types.  
Effect on original feature: Deletion of semantically well-defined feature.  
Difficulty of converting: Semantic transformation. In C++, the initializer for one of a set of mutually-referential file-local objects with static storage duration must invoke a function call to achieve the initialization.  
How widely used: Seldom.

Affected subclause: 6.4  
Change: A struct is a scope in C++, not in C.  
Rationale: Class scope is crucial to C++, and a struct is a class.  
Effect on original feature: Change to semantics of well-defined feature.  
Difficulty of converting: Semantic transformation.
How widely used: C programs use struct extremely frequently, but the change is only noticeable when struct, enumeration, or enumerator names are referred to outside the struct. The latter is probably rare.

3 Affected subclause: 6.6 [also 9.2.9]
Change: A name of file scope that is explicitly declared const, and not explicitly declared extern, has internal linkage, while in C it would have external linkage.
Rationale: Because const objects may be used as values during translation in C++, this feature urges programmers to provide an explicit initializer for each const object. This feature allows the user to put const objects in source files that are included in more than one translation unit.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom.

4 Affected subclause: 6.9.3.1
Change: The main function cannot be called recursively and cannot have its address taken.
Rationale: The main function may require special actions.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Trivial: create an intermediary function such as mymain(argc, argv).
How widely used: Seldom.

5 Affected subclause: 6.8
Change: C allows “compatible types” in several places, C++ does not.
For example, otherwise-identical struct types with different tag names are “compatible” in C but are distinctly different types in C++.
Rationale: Stricter type checking is essential for C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The “typesafe linkage” mechanism will find many, but not all, of such problems. Those problems not found by typesafe linkage will continue to function properly, according to the “layout compatibility rules” of this document.
How widely used: Common.

C.5.4 Clause 7: expressions

1 Affected subclause: 7.3.12
Change: Converting void* to a pointer-to-object type requires casting.

```c
char a[10];
void* b=a;
void foo() {
    char* c=b;
}
```

ISO C will accept this usage of pointer to void being assigned to a pointer to object type. C++ will not.
Rationale: C++ tries harder than C to enforce compile-time type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Could be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast. For example:

```c
char* c = (char*) b;
```

How widely used: This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

2 Affected subclause: 7.6.1.3
Change: Implicit declaration of functions is not allowed.
Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature. Note: the original feature was labeled as “obsolescent” in ISO C.
Difficulty of converting: Syntactic transformation. Facilities for producing explicit function declarations are fairly widespread commercially.
How widely used: Common.

3 Affected subclauses: 7.6.1.6 and 7.6.2.3
Change: Decrement operator is not allowed with bool operand.
Rationale: Feature with surprising semantics.
Effect on original feature: A valid ISO C expression utilizing the decrement operator on a `bool` lvalue (for instance, via the C typedef in `<stdbool.h>` (D.10.5)) is ill-formed in C++.

4 Affected subclauses: 7.6.2.5 and 7.6.3
Change: In C++, types can only be defined in declarations, not in expressions.
In C, a sizeof expression or cast expression may define a new type. For example,

```c
p = (void*)(struct x {int i;} *)0;
```

defines a new type, struct x.
Reason: This prohibition helps to clarify the location of definitions in the source code.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

5 Affected subclauses: 7.6.16, 7.6.19, and 7.6.20
Change: The result of a conditional expression, an assignment expression, or a comma expression may be an lvalue.
Reason: C++ is an object-oriented language, placing relatively more emphasis on lvalues. For example, function calls may yield lvalues.
Effect on original feature: Change to semantics of well-defined feature. Some C expressions that implicitly rely on lvalue-to-rvalue conversions will yield different results. For example,

```c
char arr[100];
sizeof(0, arr)
```
yields 100 in C++ and `sizeof(char*)` in C.
Difficulty of converting: Programs must add explicit casts to the appropriate rvalue.
How widely used: Rare.

C.5.5 Clause 8: statements

1 Affected subclauses: 8.5.3 and 8.7.6
Change: It is now invalid to jump past a declaration with explicit or implicit initializer (except across entire block not entered).
Reason: Constructors used in initializers may allocate resources which need to be de-allocated upon leaving the block. Allowing jump past initializers would require complicated runtime determination of allocation. Furthermore, any use of the uninitialized object could be a disaster. With this simple compile-time rule, C++ assures that if an initialized variable is in scope, then it has assuredly been initialized.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom.

2 Affected subclause: 8.7.4
Change: It is now invalid to return (explicitly or implicitly) from a function which is declared to return a value without actually returning a value.
Reason: The caller and callee may assume fairly elaborate return-value mechanisms for the return of class objects. If some flow paths execute a return without specifying any value, the implementation must embody many more complications. Besides, promising to return a value of a given type, and then not returning such a value, has always been recognized to be a questionable practice, tolerated only because very-old C had no distinction between void functions and int functions.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Add an appropriate return value to the source code, such as zero.
How widely used: Seldom. For several years, many existing C implementations have produced warnings in this case.

C.5.6 Clause 9: declarations

1 Affected subclause: 9.2.2
Change: In C++, the static or extern specifiers can only be applied to names of objects or functions. Using these specifiers with type declarations is illegal in C++. In C, these specifiers are ignored when used on type declarations.

Example:
static struct S { // valid C, invalid in C++
    int i;
};

Rationale: Storage class specifiers don’t have any meaning when associated with a type. In C++, class members can be declared with the `static` storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation.

How widely used: Seldom.

2 Affected subclause: 9.2.2
Change: In C++, `register` is not a storage class specifier.
Rationale: The storage class specifier had no effect in C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Common.

3 Affected subclause: 9.2.4
Change: A C++ typedef name must be different from any class type name declared in the same scope (except if the typedef is a synonym of the class name with the same name). In C, a typedef name and a struct tag name declared in the same scope can have the same name (because they have different name spaces).

Example:

```c
typedef struct name1 { /* ... */ } name1; // valid C and C++
struct name { /* ... */ };            // valid C, invalid C++
typedef int name;                    // valid C, invalid C++
```

Rationale: For ease of use, C++ doesn’t require that a type name be prefixed with the keywords `class`, `struct` or `union` when used in object declarations or type casts.

Example:

```c
class name { /* ... */ };            // i has type `class name`
nome i;                            // i has type `class name`
```

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. One of the 2 types has to be renamed.

How widely used: Seldom.

4 Affected subclause: 9.2.9 [see also 6.6]
Change: Const objects must be initialized in C++ but can be left uninitialized in C.
Rationale: A const object cannot be assigned to so it must be initialized to hold a useful value.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom.

5 Affected subclause: 9.2.9
Change: Banning implicit `int`.

In C++ a `decl-specifier-seq` must contain a `type-specifier`, unless it is followed by a declarator for a constructor, a destructor, or a conversion function. In the following example, the left-hand column presents valid C; the right-hand column presents equivalent C++:

```c
void f(const parm); void f(const int parm);
const n = 3; const int n = 3;
main() int main()
    /* ... */ /* ... */
```

Rationale: In C++, implicit int creates several opportunities for ambiguity between expressions involving function-like casts and declarations. Explicit declaration is increasingly considered to be proper style. Liaison with WG14 (C) indicated support for (at least) deprecating implicit int in the next revision of C.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation. Could be automated.

How widely used: Common.

6 Affected subclause: 9.2.9.6
Change: The keyword `auto` cannot be used as a storage class specifier.
void f() {
    auto int x;  // valid C, invalid C++
}

Rationale: Allowing the use of auto to deduce the type of a variable from its initializer results in undesired interpretations of auto as a storage class specifier in certain contexts.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Rare.

7 Affected subclause: 9.3.4.6
Change: In C++, a function declared with an empty parameter list takes no arguments. In C, an empty parameter list means that the number and type of the function arguments are unknown.
Example:
int f();  // means int f(void) in C++
    // int f( unknown ) in C
Rationale: This is to avoid erroneous function calls (i.e., function calls with the wrong number or type of arguments).
Effect on original feature: Change to semantics of well-defined feature. This feature was marked as “obsolescent” in C.
Difficulty of converting: Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.
How widely used: Common.

8 Affected subclause: 9.3.4.6 [see 7.6.2.5]
Change: In C++, types may not be defined in return or parameter types. In C, these type definitions are allowed.
Example:
void f( struct S { int a; } arg ) {}  // valid C, invalid C++
enum E { A, B, C } f() {}  // valid C, invalid C++
Rationale: When comparing types in different translation units, C++ relies on name equivalence when C relies on structural equivalence. Regarding parameter types: since the type defined in a parameter list would be in the scope of the function, the only legal calls in C++ would be from within the function itself.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The type definitions must be moved to file scope, or in header files.
How widely used: Seldom. This style of type definition is seen as poor coding style.

9 Affected subclause: 9.5
Change: In C++, the syntax for function definition excludes the “old-style” C function. In C, “old-style” syntax is allowed, but deprecated as “obsolescent”.
Rationale: Prototypes are essential to type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Common in old programs, but already known to be obsolescent.

10 Affected subclause: 9.4.2
Change: In C++, designated initialization support is restricted compared to the corresponding functionality in C. In C++, designators for non-static data members must be specified in declaration order, designators for array elements and nested designators are not supported, and designated and non-designated initializers cannot be mixed in the same initializer list.
Example:
struct A { int x, y; };
struct B { struct A a; };
struct A a = {.y = 1, .x = 2};  // valid C, invalid C++
int arr[3] = {{11} = 5};  // valid C, invalid C++
struct B b = {.a.x = 0};  // valid C, invalid C++
struct A c = {.x = 1, 2};  // valid C, invalid C++
Rationale: In C++, members are destroyed in reverse construction order and the elements of an initializer list are evaluated in lexical order, so field initializers must be specified in order. Array designators conflict with lambda-expression syntax. Nested designators are seldom used.

Effect on original feature: Deletion of feature that is incompatible with C++.

Difficulty of converting: Syntactic transformation.

How widely used: Out-of-order initializers are common. The other features are seldom used.

Affected subclause: 9.4.3

Change: In C++, when initializing an array of character with a string, the number of characters in the string (including the terminating ‘\0’) must not exceed the number of elements in the array. In C, an array can be initialized with a string even if the array is not large enough to contain the string-terminating ‘\0’.

Example:
```c
char array[4] = "abcd"; // valid C, invalid C++
```

Rationale: When these non-terminated arrays are manipulated by standard string functions, there is potential for major catastrophe.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. The arrays must be declared one element bigger to contain the string terminating ‘\0’.

How widely used: Seldom. This style of array initialization is seen as poor coding style.

Affected subclause: 9.7.1

Change: C++ objects of enumeration type can only be assigned values of the same enumeration type. In C, objects of enumeration type can be assigned values of any integral type.

Example:
```c
enum color { red, blue, green };  // valid C, invalid C++
enum color c = 1;  // valid C, invalid C++
```

Rationale: The type-safe nature of C++.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast.)

How widely used: Common.

Affected subclause: 9.7.1

Change: In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is int.

Example:
```c
enum e { A };  // in C
sizeof(A) == sizeof(int)  // in C
sizeof(A) == sizeof(e)  // in C++
/* and sizeof(int) is not necessarily equal to sizeof(e) */
```

Rationale: In C++, an enumeration is a distinct type.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation.

How widely used: Seldom. The only time this exists affecting existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

C.5.7 Clause 11: classes [diff.class]
Rationale: This is one of the few incompatibilities between C and C++ that can be attributed to the new C++ name space definition where a name can be declared as a type and as a non-type in a single scope causing the non-type name to hide the type name and requiring that the keywords class, struct, union or enum be used to refer to the type name. This new name space definition provides important notational conveniences to C++ programmers and helps making the use of the user-defined types as similar as possible to the use of fundamental types. The advantages of the new name space definition were judged to outweigh by far the incompatibility with C described above.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation. If the hidden name that needs to be accessed is at global scope, the :: C++ operator can be used. If the hidden name is at block scope, either the type or the struct tag has to be renamed.

How widely used: Seldom.

Affected subclause: 11.4.5.3

Change: Copying volatile objects.

The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

```c
struct X { int i; };  
volatile struct X x1 = {0};  
struct X x2 = x1;  // invalid C++  
struct X x3;  
x3 = x1;  // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to volatile const X& would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. If volatile semantics are required for the copy, a user-declared constructor or assignment must be provided. If non-volatile semantics are required, an explicit const_cast can be used.

How widely used: Seldom.

Affected subclause: 11.4.10

Change: Bit-fields of type plain int are signed.

Rationale: Leaving the choice of signedness to implementations could lead to inconsistent definitions of template specializations. For consistency, the implementation freedom was eliminated for non-dependent types, too.

Effect on original feature: The choice is implementation-defined in C, but not so in C++.

Difficulty of converting: Syntactic transformation.

How widely used: Seldom.

Affected subclause: 11.4.11

Change: In C++, the name of a nested class is local to its enclosing class. In C the name of the nested class belongs to the same scope as the name of the outermost enclosing class.

Example:

```c
struct X {  
    struct Y { /* ... */ } y;  
};  
struct Y yy;  // valid C, invalid C++
```

Rationale: C++ classes have member functions which require that classes establish scopes. The C rule would leave classes as an incomplete scope mechanism which would prevent C++ programmers from maintaining locality within a class. A coherent set of scope rules for C++ based on the C rule would be very complicated and C++ programmers would be unable to predict reliably the meanings of nontrivial examples involving nested or local functions.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:
struct Y;                    // struct Y and struct X are at the same scope
struct X {
    struct Y { /* ... */ } y;
};

All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct. Note: this is a consequence of the difference in scope rules, which is documented in 6.4.

How widely used: Seldom.

Affected subclause: 11.4.12

Change: In C++, a typedef name may not be redeclared in a class definition after being used in that definition.

Example:

typedef int I;
struct S {
    I i;
    int I;            // valid C, invalid C++
};

Rationale: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of I really is.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. Either the type or the struct member has to be renamed.

How widely used: Seldom.

C.5.8 Clause 15: preprocessing directives

Affected subclause: 15.11

Change: Whether __STDC__ is defined and if so, what its value is, are implementation-defined.

Rationale: C++ is not identical to ISO C. Mandating that __STDC__ be defined would require that translators make an incorrect claim.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation.

How widely used: Programs and headers that reference __STDC__ are quite common.

C.6 C standard library

C.6.1 General

Subclause C.6 summarizes the explicit changes in headers, definitions, declarations, or behavior between the C standard library in the C standard and the parts of the C++ standard library that were included from the C standard library.

C.6.2 Modifications to headers

For compatibility with the C standard library, the C++ standard library provides the C headers enumerated in D.10, but their use is deprecated in C++.

There are no C++ headers for the C standard library's headers <stdatomic.h>, <stdnoreturn.h>, and <threads.h>, nor are these headers from the C standard library headers themselves part of C++.

The C headers <complex.h> and <tgmath.h> do not contain any of the content from the C standard library and instead merely include other headers from the C++ standard library.

C.6.3 Modifications to definitions

The types char16_t and char32_t are distinct types rather than typedefs to existing integral types. The tokens char16_t and char32_t are keywords in C++ (5.11). They do not appear as macro or type names defined in <cuchar> (21.5.5).
C.6.3.2 Type wchar_t

The type wchar_t is a distinct type rather than a typedef to an existing integral type. The token wchar_t is a keyword in C++ (5.11). It does not appear as a macro or type name defined in any of <cstdlib> (17.2.2), <cwchar> (21.5.4).

C.6.3.3 Header <cassert.h>

The token static_assert is a keyword in C++. It does not appear as a macro name defined in <cassert> (19.3.2).

C.6.3.4 Header <iso646.h>

The tokens and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, and xor_eq are keywords in C++ (5.11), and are not introduced as macros by <iso646.h> (D.10.3).

C.6.3.5 Header <stdalign.h>

The token alignas is a keyword in C++ (5.11), and is not introduced as a macro by <stdalign.h> (D.10.4).

C.6.3.6 Header <stdbool.h>

The tokens bool, true, and false are keywords in C++ (5.11), and are not introduced as macros by <stdbool.h> (D.10.5).

C.6.3.7 Macro NULL

The macro NULL, defined in any of <locale> (28.5.1), <cstddef> (17.2.1), <cstdio> (29.12.1), <cstdlib> (17.2.2), <cstring> (21.5.3), <ctime> (27.14), or <cwchar> (21.5.4), is an implementation-defined null pointer constant in C++ (17.2).

C.6.4 Modifications to declarations

1 Header <cstring> (21.5.3): The following functions have different declarations:

- strchr
- strpbrk
- strrchr
- strstr
- memchr

Subclause 21.5.3 describes the changes.

2 Header <cwchar> (21.5.4): The following functions have different declarations:

- wcschr
- wcpbrk
- wcsrchr
- wcsstr
- wmemchr

Subclause 21.5.4 describes the changes.

3 Header <cstdlib> (17.2.1) declares the names nullptr_t, byte, and to_integer, and the operators and operator templates in (17.2.5), in addition to the names declared in <stddef.h> (D.10) in the C standard library.

C.6.5 Modifications to behavior

C.6.5.1 General

1 Header <cstdlib> (17.2.2): The following functions have different behavior:

- atexit
- exit
- abort
Subclause 17.5 describes the changes.

2 Header `<csetjmp>` (17.13.3): The following functions have different behavior:

(2.1) — `longjmp`

Subclause 17.13.3 describes the changes.

C.6.5.2 Macro `offsetof(type, member-designator)`

1 The macro `offsetof`, defined in `<cstddef>` (17.2.1), accepts a restricted set of `type` arguments in C++. Subclause 17.2.4 describes the change.

C.6.5.3 Memory allocation functions

1 The functions `aligned_alloc`, `calloc`, `malloc`, and `realloc` are restricted in C++. Subclause 20.10.12 describes the changes.
Annex D  (normative)
Compatibility features  [depr]

D.1 General  [depr.general]
1 This Annex describes features of the C++ Standard that are specified for compatibility with existing implementations.
2 These are deprecated features, where deprecated is defined as: Normative for the current revision of C++, but having been identified as a candidate for removal from future revisions. An implementation may declare library names and entities described in this Clause with the deprecated attribute (9.12.4).

D.2 Arithmetic conversion on enumerations  [depr.arith.conv.enum]
1 The ability to apply the usual arithmetic conversions (7.4) on operands where one is of enumeration type and the other is of a different enumeration type or a floating-point type is deprecated.

Example 1:
```cpp
enum E1 { e };
enum E2 { f };
bool b = e <= 3.7;     // deprecated
int k = f - e;         // deprecated
auto cmp = e <=> f;    // error
```

D.3 Implicit capture of *this by reference  [depr.capture.this]
1 For compatibility with prior revisions of C++, a lambda-expression with capture-default = (7.5.5.3) may implicitly capture *this by reference.

Example 1:
```cpp
struct X {
    int x;
    void foo(int n) {
        auto f = [=]() { x = n; };     // deprecated: x means this->x, not a copy thereof
        auto g = [=, this]() { x = n; }; // recommended replacement
    }
};
```

D.4 Comma operator in subscript expressions  [depr.comma.subscript]
1 A comma expression (7.6.20) appearing as the expr-or-braced-init-list of a subscripting expression (7.6.1.2) is deprecated.

Note 1: A parenthesized comma expression is not deprecated. — end note

Example 1:
```cpp
void f(int *a, int b, int c) {
    a[b,c];                     // deprecated
    a[(b,c)];                   // OK
}
```

D.5 Array comparisons  [depr.array.comp]
1 Equality and relational comparisons (7.6.10, 7.6.9) between two operands of array type are deprecated.

Note 1: Three-way comparisons (7.6.8) between such operands are ill-formed. — end note

Example 1:
int arr1[5];
int arr2[5];
bool same = arr1 == arr2;  // deprecated, same as &arr1[0] == &arr2[0],
                          // does not compare array contents
auto cmp = arr1 <=> arr2; // error
—end example]

D.6 Deprecated volatile types

1 Postfix ++ and -- expressions (7.6.1.6) and prefix ++ and -- expressions (7.6.2.3) of volatile-qualified arithmetic and pointer types are deprecated.
[Example 1:
  volatile int velociraptor;
  ++velociraptor;  // deprecated
—end example]

2 Certain assignments where the left operand is a volatile-qualified non-class type are deprecated; see 7.6.19.
[Example 2:
  int neck, tail;
  volatile int brachiosaur;
  brachiosaur = neck;  // OK
  tail = brachiosaur;  // OK
  tail = brachiosaur = neck;  // deprecated
  brachiosaur += neck;  // deprecated
  brachiosaur = brachiosaur + neck;  // OK
—end example]

3 A function type (9.3.4.6) with a parameter with volatile-qualified type or with a volatile-qualified return type is deprecated.
[Example 3:
  volatile struct amber jurassic();  // deprecated
  void trex(volatile short left_arm, volatile short right_arm);  // deprecated
  void fly(volatile struct pterosaur* pteranodon);  // OK
—end example]

4 A structured binding (9.6) of a volatile-qualified type is deprecated.
[Example 4:
  struct linhenykus { short forelimb; }
  void park(linhenykus alvarezsauroid) {
    volatile auto [what_is_this] = alvarezsauroid;    // deprecated
  }
—end example]

D.7 Redeclaration of static constexpr data members

1 For compatibility with prior revisions of C++, a constexpr static data member may be redundantly redeclared outside the class with no initializer. This usage is deprecated.
[Example 1:
  struct A {
    static constexpr int n = 5;  // definition (declaration in C++ 2014)
  };
  constexpr int A::n;  // redundant declaration (definition in C++ 2014)
—end example]

D.8 Non-local use of TU-local entities

1 A declaration of a non-TU-local entity that is an exposure (6.6) is deprecated.
[Note 1: Such a declaration in an importable module unit is ill-formed. — end note]
Example 1:

```cpp
namespace {
    struct A {
        void f() {}
    };
}
A h(); // deprecated: not internal linkage
inline void g() {A().f();} // deprecated: inline and not internal linkage
```

--- end example

D.9 Implicit declaration of copy functions

The implicit definition of a copy constructor (11.4.5.3) as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor (11.4.7). The implicit definition of a copy assignment operator (11.4.6) as defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor. In a future revision of C++, these implicit definitions could become deleted (9.5.3).

D.10 C headers

D.10.1 General

For compatibility with the C standard library, the C++ standard library provides the C headers shown in Table 147.

Table 147: C headers

<table>
<thead>
<tr>
<th>Header</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;assert.h&gt;</code></td>
<td><code>&lt;inttypes.h&gt;</code></td>
</tr>
<tr>
<td><code>&lt;complex.h&gt;</code></td>
<td><code>&lt;iso646.h&gt;</code></td>
</tr>
<tr>
<td><code>&lt;ctype.h&gt;</code></td>
<td><code>&lt;limits.h&gt;</code></td>
</tr>
<tr>
<td><code>&lt;errno.h&gt;</code></td>
<td><code>&lt;stdarg.h&gt;</code></td>
</tr>
<tr>
<td><code>&lt;float.h&gt;</code></td>
<td><code>&lt;setjmp.h&gt;</code></td>
</tr>
</tbody>
</table>

D.10.2 Header `<complex.h>` synopsis

The header `<complex.h>` behaves as if it simply includes the header `<complex>` (26.4.2).

[Note 1: Names introduced by `<complex>` in namespace `std` are not placed into the global namespace scope by `<complex.h>`. — end note]

D.10.3 Header `<iso646.h>` synopsis

The C++ header `<iso646.h>` is empty.

[Note 1: `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor`, and `xor_eq` are keywords in C++ (5.11). — end note]

D.10.4 Header `<stdalign.h>` synopsis

The contents of the C++ header `<stdalign.h>` are the same as the C standard library header `<stdalign.h>`.

See also: ISO C 7.15

D.10.5 Header `<stdbool.h>` synopsis

The contents of the C++ header `<stdbool.h>` are the same as the C standard library header `<stdbool.h>`.

See also: ISO C 7.18
D.10.6 Header `<tgmath.h>` synopsis

```c
#include <cmath>
#include <complex>
```

1 The header `<tgmath.h>` behaves as if it simply includes the headers `<cmath>` (26.8.1) and `<complex>` (26.4.2).

2 [Note 1: The overloads provided in C by type-generic macros are already provided in `<complex>` and `<cmath>` by “sufficient” additional overloads. — end note]

3 [Note 2: Names introduced by `<cmath>` or `<complex>` in namespace `std` are not placed into the global namespace scope by `<tgmath.h>`. — end note]

D.10.7 Other C headers

1 Every C header other than `<complex.h>` (D.10.2), `<iso646.h>` (D.10.3), `<stdalign.h>` (D.10.4), `<stdbool.h>` (D.10.5), and `<tgmath.h>` (D.10.6), each of which has a name of the form `<name>.h>`, behaves as if each name placed in the standard library namespace by the corresponding `<cname>` header is placed within the global namespace scope, except for the functions described in 26.8.6, the declaration of `std::byte` (17.2.1), and the functions and function templates described in 17.2.5. It is unspecified whether these names are first declared or defined within namespace scope (6.4.6) of the namespace `std` and are then injected into the global namespace scope by explicit using-declarations (9.9).

2 [Example 1: The header `<cstdlib>` assuredly provides its declarations and definitions within the namespace `std`. It may also provide these names within the global namespace. The header `<stdlib.h>` assuredly provides the same declarations and definitions within the global namespace, much as in the C Standard. It may also provide these names within the namespace `std`. — end example]

D.11 Requires paragraph

In addition to the elements specified in 16.3.2.4, descriptions of function semantics may also contain a Requires: element to denote the preconditions for calling a function.

2 Violation of any preconditions specified in a function’s Requires: element results in undefined behavior unless the function’s Throws: element specifies throwing an exception when the precondition is violated.

D.12 Relational operators

1 The header `<utility>` (20.2.1) has the following additions:

```c
namespace std::rel_ops {
    template<class T> bool operator!=(const T&, const T&);
    template<class T> bool operator>(const T& x, const T& y);
    template<class T> bool operator<=(const T& x, const T& y);
    template<class T> bool operator>=(const T& x, const T& y);
}
```

2 To avoid redundant definitions of `operator!=` out of `operator==` and operators `>`, `<=`, and `>=` out of `operator<`, the library provides the following:

```c
template<class T> bool operator!=(const T& x, const T& y);
```

3 Requires: Type `T` is `Cpp17EqualityComparable` (Table 25).

4 Returns: `!(x == y)`.

```c
template<class T> bool operator>(const T& x, const T& y);
```

5 Requires: Type `T` is `Cpp17LessThanComparable` (Table 26).

6 Returns: `y < x`.

```c
template<class T> bool operator<=(const T& x, const T& y);
```

7 Requires: Type `T` is `Cpp17LessThanComparable` (Table 26).

8 Returns: `!(y < x)`.

```c
template<class T> bool operator>=(const T& x, const T& y);
```

9 Requires: Type `T` is `Cpp17LessThanComparable` (Table 26).

10 Returns: `!(x < y)`.
The class `strstreambuf` associates the input sequence, and possibly the output sequence, with an object of some `character` array type, whose elements store arbitrary values. The array object has several attributes.

1. The class `strstreambuf` associates the input sequence, and possibly the output sequence, with an object of some `character` array type, whose elements store arbitrary values. The array object has several attributes.

2. [Note 1: For the sake of exposition, these are represented as elements of a bitmask type (indicated here as `T1`) called `strstate`. The elements are:]
— allocated, set when a dynamic array object has been allocated, and hence will be freed by the destructor for the `strstreambuf` object;

— constant, set when the array object has `const` elements, so the output sequence cannot be written;

— dynamic, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;

— frozen, set when the program has requested that the array object not be altered, reallocated, or freed.

—end note

[Note 2: For the sake of exposition, the maintained data is presented here as:

—(3.1) `strstate strmode`, the attributes of the array object associated with the `strstreambuf` object;

—(3.2) `int alsize`, the suggested minimum size for a dynamic array object;

—(3.3) `void* (*palloc)(size_t)`, points to the function to call to allocate a dynamic array object;

—(3.4) `void (*pfree)(void*)`, points to the function to call to free a dynamic array object.

—end note]

Each object of class `strstreambuf` has a seekable area, delimited by the pointers `seeklow` and `seekhigh`. If `gnext` is a null pointer, the seekable area is undefined. Otherwise, `seeklow` equals `gbeg` and `seekhigh` is either `pend`, if `pend` is not a null pointer, or `gend`.

D.13.2.2 strstreambuf constructors

`explicit strstreambuf(streamsize alsize_arg);`

Effects: Initializes the base class with `streambuf()`. The postconditions of this function are indicated in Table 148.

```
<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>dynamic</td>
</tr>
<tr>
<td>alsize</td>
<td>alsize_arg</td>
</tr>
<tr>
<td>palloc</td>
<td>a null pointer</td>
</tr>
<tr>
<td>pfree</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>
```

`strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));`

Effects: Initializes the base class with `streambuf()`. The postconditions of this function are indicated in Table 149.

```
<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>dynamic</td>
</tr>
<tr>
<td>alsize</td>
<td>an unspecified value</td>
</tr>
<tr>
<td>palloc</td>
<td>palloc_arg</td>
</tr>
<tr>
<td>pfree</td>
<td>pfree_arg</td>
</tr>
</tbody>
</table>
```

`strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = nullptr);`

`strstreambuf(signed char* gnext_arg, streamsize n, signed char* pbeg_arg = nullptr);`

`strstreambuf(unsigned char* gnext_arg, streamsize n, unsigned char* pbeg_arg = nullptr);`

Effects: Initializes the base class with `streambuf()`. The postconditions of this function are indicated in Table 150.

`gnext_arg` shall point to the first element of an array object whose number of elements `N` is determined as follows:
Table 150: `strstreambuf(charT*, streamsize, charT*)` effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>0</td>
</tr>
<tr>
<td>alsize</td>
<td>an unspecified value</td>
</tr>
<tr>
<td>palloc</td>
<td>a null pointer</td>
</tr>
<tr>
<td>pfree</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

(4.1) If \(n > 0\), \(N\) is \(n\).
(4.2) If \(n == 0\), \(N\) is `std::strlen(gnext_arg)`.
(4.3) If \(n < 0\), \(N\) is `INT_MAX`.

If `pbeg_arg` is a null pointer, the function executes:
```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```
Otherwise, the function executes:
```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg, pbeg_arg + N);
```

\[\text{strstreambuf(const char* gnext_arg, streamsize n);}\]
\[\text{strstreambuf(const signed char* gnext_arg, streamsize n);}\]
\[\text{strstreambuf(const unsigned char* gnext_arg, streamsize n);}\]

7 Effects: Behaves the same as `strstreambuf((char*)gnext_arg, n)`, except that the constructor also sets constant in `strmode`.

```
virtual ~strstreambuf();
```

8 Effects: Destroys an object of class `strstreambuf`. The function frees the dynamically allocated array object only if `(strmode & allocated) != 0` and `(strmode & frozen) == 0`. (D.13.2.4 describes how a dynamically allocated array object is freed.)

D.13.2.3 Member functions

```
void freeze(bool freezefl = true);
```

1 Effects: If `strmode & dynamic` is nonzero, alters the freeze status of the dynamic array object as follows:

(1.1) If `freezefl` is `true`, the function sets `frozen` in `strmode`.
(1.2) Otherwise, it clears `frozen` in `strmode`.

```
char* str();
```

2 Effects: Calls `freeze()`, then returns the beginning pointer for the input sequence, `gbeg`.

Remarks: The return value can be a null pointer.

```
int pcount() const;
```

4 Effects: If the next pointer for the output sequence, `pnext`, is a null pointer, returns zero. Otherwise, returns the current effective length of the array object as the next pointer minus the beginning pointer for the output sequence, `pnext - pbeg`.

D.13.2.4 `strstreambuf` overridden virtual functions

```
int_type overflow(int_type c = EOF) override;
```

1 Effects: Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

(1.1) If `c` != `EOF` and if either the output sequence has a write position available or the function makes a write position available (as described below), assigns `c` to `*pnext++`.

Returns `(unsigned char)c`.

---

330) The function signature `strlen(const char*)` is declared in `<cstring>` (21.5.3). The macro `INT_MAX` is defined in `<climits>` (17.3.6).
If \( c == EOF \), there is no character to append.

Returns a value other than EOF.

Returns EOF to indicate failure.

Remarks: The function can alter the number of write positions available as a result of any call.

To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements \( n \) to hold the current array object (if any), plus at least one additional write position. How many additional write positions are made available is otherwise unspecified. If \( \text{palloc} \) is not a null pointer, the function calls \((\text{palloc})(n)\) to allocate the new dynamic array object. Otherwise, it evaluates the expression \( \text{new} \) charT\[n\]. In either case, if the allocation fails, the function returns EOF. Otherwise, it sets \( \text{allocated} \) in \( \text{strmode} \).

To free a previously existing dynamic array object whose first element address is \( p \): If \( \text{pfree} \) is not a null pointer, the function calls \((\text{pfree})(p)\). Otherwise, it evaluates the expression \( \text{delete}[]p \).

If (\( \text{strmode} & \text{dynamic} \) == 0, or if (\( \text{strmode} & \text{frozen} \) != 0, the function cannot extend the array (reallocating it with greater length) to make a write position available.

Recommended practice: An implementation should consider \( \text{alsize} \) in making the decision how many additional write positions to make available.

\[
\text{int\_type pbackfail(int\_type c = EOF) override;}
\]

Puts back the character designated by \( c \) to the input sequence, if possible, in one of three ways:

(8.1) If \( c != EOF \), if the input sequence has a putback position available, and if (char)\( c == \text{gnext}[-1] \), assigns \( \text{gnext} - 1 \) to \( \text{gnext} \).

Returns \( c \).

(8.2) If \( c != EOF \), if the input sequence has a putback position available, and if \( \text{strmode} & \text{constant} \) is zero, assigns \( c \) to \(*--\text{gnext} \).

Returns \( c \).

(8.3) If \( c == EOF \) and if the input sequence has a putback position available, assigns \( \text{gnext} - 1 \) to \( \text{gnext} \).

Returns a value other than EOF.

Returns EOF to indicate failure.

Remarks: If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

\[
\text{int\_type underflow() override;}
\]

Effects: Reads a character from the input sequence, if possible, without moving the stream position past it, as follows:

(11.1) If the input sequence has a read position available, the function signals success by returning (unsigned char)*\( \text{gnext} \).

(11.2) Otherwise, if the current write next pointer \( \text{pnext} \) is not a null pointer and is greater than the current read end pointer \( \text{gend} \), makes a read position available by assigning to \( \text{gend} \) a value greater than \( \text{gnext} \) and no greater than \( \text{pnext} \).

Returns (unsigned char)*\( \text{gnext} \).

Returns EOF to indicate failure.

Remarks: The function can alter the number of read positions available as a result of any call.

\[
\text{pos\_type seekoff(off\_type off, seekdir way, openmode which = in | out) override;}
\]

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 151.

For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines \( \text{newoff} \) as indicated in Table 152.

If \( \text{newoff} + \text{off} < (\text{seeklow} - \text{xbeg}) \) or \( \text{seekhigh} - \text{xbeg} < (\text{newoff} + \text{off}) \), the positioning operation fails. Otherwise, the function assigns \( \text{xbeg} + \text{newoff} + \text{off} \) to the next pointer \( \text{xnext} \).
Table 151: seekoff positioning

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(which &amp; ios::in) != 0</td>
<td>positions the input sequence</td>
</tr>
<tr>
<td>(which &amp; ios::out) != 0</td>
<td>positions the output sequence</td>
</tr>
<tr>
<td>(which &amp; (ios::in</td>
<td>ios::out)) == (ios::in</td>
</tr>
<tr>
<td>Otherwise</td>
<td>the positioning operation fails.</td>
</tr>
</tbody>
</table>

Table 152: newoff values

<table>
<thead>
<tr>
<th>Condition</th>
<th>newoff Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>way == ios::beg</td>
<td>0</td>
</tr>
<tr>
<td>way == ios::cur</td>
<td>the next pointer minus the beginning pointer (xnext - xbeg).</td>
</tr>
<tr>
<td>way == ios::end</td>
<td>seekhigh minus the beginning pointer (seekhigh - xbeg).</td>
</tr>
</tbody>
</table>

Returns: pos_type(newoff), constructed from the resultant offset newoff (of type off_type), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is pos_type(off_type(-1)).

pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out) override;

Effects: Alters the stream position within one of the controlled sequences, if possible, to correspond to the stream position stored in sp (as described below).

(18.1) — If (which & ios::in) != 0, positions the input sequence.
(18.2) — If (which & ios::out) != 0, positions the output sequence.
(18.3) — If the function positions neither sequence, the positioning operation fails.

For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines newoff from sp.offset():

(19.1) — If newoff is an invalid stream position, has a negative value, or has a value greater than (seekhigh - seeklow), the positioning operation fails
(19.2) — Otherwise, the function adds newoff to the beginning pointer xbeg and stores the result in the next pointer xnext.

Returns: pos_type(newoff), constructed from the resultant offset newoff (of type off_type), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is pos_type(off_type(-1)).

streambuf<char*> setbuf(char* s, streamsize n) override;

Effects: Behavior is implementation-defined, except that setbuf(0, 0) has no effect.

D.13.3 Class istrstream

D.13.3.1 General

namespace std {

class istrstream : public basic_istream<char> {
public:
explicit istrstream(const char* s);
explicit istrstream(char* s);
istrstream(const char* s, streamsize n);
istrstream(char* s, streamsize n);
virtual ~istrstream();

§ D.13.3.1 1689
The class `istrstream` supports the reading of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

\[ (1.1) \quad \text{sb}, \text{the strstreambuf object.} \]

### D.13.3.2 `istream` constructors

```cpp
explicit istrstream(const char* s);
explicit istrstream(char* s);
```

1. **Effects:** Initializes the base class with `istream(&sb)` and `sb` with `strstreambuf(s, 0)`. `s` shall designate the first element of an `ntbs`.

```cpp
istrstream(const char* s, streamsize n);
istrstream(char* s, streamsize n);
```

2. **Effects:** Initializes the base class with `istream(&sb)` and `sb` with `strstreambuf(s, n)`. `s` shall designate the first element of an array whose length is `n` elements, and `n` shall be greater than zero.

### D.13.3.3 Member functions

```cpp
strstreambuf* rdbuf() const;  
char* str();
```

1. **Returns:** `const_cast<strstreambuf*>(&sb)`.

2. **Returns:** `rdbuf()->str()`.

### D.13.4 Class `ostrstream` ([depr.ostrstream]

The class `ostrstream` supports the writing of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

\[ (1.1) \quad \text{sb}, \text{the strstreambuf object.} \]

### D.13.4.2 `ostrstream` constructors

```cpp
ostrstream();
```

1. **Effects:** Initializes the base class with `ostream(&sb)` and `sb` with `strstreambuf()`.
ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);

Effects: Initializes the base class with ostream(&sb), and sb with one of two constructors:

(2.1) If (mode & app) == 0, then s shall designate the first element of an array of n elements.
    The constructor is strstreambuf(s, n, s).

(2.2) If (mode & app) != 0, then s shall designate the first element of an array of n elements that
    contains an nTBS whose first element is designated by s. The constructor is strstreambuf(s, n,
    s + std::strlen(s)).

D.13.4.3 Member functions

strstreambuf* rdbuf() const;

Returns: (strstreambuf*)&sb.

Effects: Calls rdbuf()->freeze(freezefl).

char* str();

Returns: rdbuf()->str().

int pcount() const;

Returns: rdbuf()->pcount().

D.13.5 Class strstream

D.13.5.1 General

namespace std {

    class strstream : public basic_iostream<char> {

    public:
        // types
        using char_type = char;
        using int_type = char_traits<char>::int_type;
        using pos_type = char_traits<char>::pos_type;
        using off_type = char_traits<char>::off_type;

        // constructors/destructor
        strstream();
        strstream(char* s, int n,
                ios_base::openmode mode = ios_base::in|ios_base::out);
        virtual ~strstream();

        // members
        strstreambuf* rdbuf() const;
        void freeze(bool freezefl = true);
        int pcount() const;
        char* str();

    private:
        strstreambuf sb; // exposition only
    };

}

1 The class strstream supports reading and writing from objects of class strstreambuf. It supplies a
strstreambuf object to control the associated array object. For the sake of exposition, the maintained data
is presented here as:

(1.1) — sb, the strstreambuf object.

331) The function signature strlen(const char*) is declared in <cstring> (21.5.3).
D.13.5.2 **strstream constructors**

`strstream();`

*Effects:* Initializes the base class with `iostream(&sb)`.

`strstream(char* s, int n, ios_base::openmode mode = ios_base::in|ios_base::out);`

*Effects:* Initializes the base class with `iostream(&sb)` and `sb` with one of the two constructors:

1. If `(mode & app) == 0`, then `s` shall designate the first element of an array of `n` elements. The constructor is `strstreambuf(s,n,s)`.
2. If `(mode & app) != 0`, then `s` shall designate the first element of an array of `n` elements that contains an `NTBS` whose first element is designated by `s`. The constructor is `strstreambuf(s,n,s + std::strlen(s))`.

D.13.5.3 **strstream destructor**

`virtual ~strstream();`

*Effects:* Destroys an object of class `strstream`.

D.13.5.4 **strstream operations**

`strstreambuf* rdbuf() const;`

*Returns:* `const_cast<strstreambuf*>(&sb)`.

`void freeze(bool freezefl = true);`

*Effects:* Calls `rdbuf()->freeze(freezefl)`.

`char* str();`

*Returns:* `rdbuf()->str()`.

`int pcount() const;`

*Returns:* `rdbuf()->pcount()`.

D.14 **Deprecated type traits**

The header `<type_traits>` (20.15.3) has the following addition:

```cpp
namespace std {
    template<class T> struct is_pod;
    template<class T> inline constexpr bool is_pod_v = is_pod<T>::value;
}
```

*Requires:* `remove_all_extents_t<T>` shall be a complete type or `cv void`.

`is_pod<T>` is a `Cpp17UnaryTypeTrait` (20.15.2) with a base characteristic of `true_type` if `T` is a POD type, and `false_type` otherwise. A POD class is a class that is both a trivial class and a standard-layout class, and has no non-static data members of type non-POD class (or array thereof). A POD type is a scalar type, a POD class, an array of such a type, or a `cv`-qualified version of one of these types.

[Note 1: It is unspecified whether a closure type (7.5.5.2) is a POD type. — end note]

D.15 **Tuple**

The header `<tuple>` (20.5.2) has the following additions:

```cpp
namespace std {
    template<class T> class tuple_size<volatile T>;
    template<class T> class tuple_size<const volatile T>;
}
```
template<size_t I, class T> class tuple_element<I, volatile T>;
template<size_t I, class T> class tuple_element<I, const volatile T>;
}

template<class T> class tuple_size<volatile T>;
template<class T> class tuple_size<const volatile T>;

Let TS denote tuple_size<T> of the cv-unqualified type T. If the expression TS::value is well-formed when treated as an unevaluated operand, then specializations of each of the two templates meet the Cpp17TransformationTrait requirements with a base characteristic of integral_constant<size_t, TS::value>. Otherwise, they have no member value.

Access checking is performed as if in a context unrelated to TS and T. Only the validity of the immediate context of the expression is considered.

In addition to being available via inclusion of the <tuple> (20.5.2) header, the two templates are available when any of the headers <array> (22.3.2), <ranges> (24.2), or <utility> (20.2.1) are included.

template<size_t I, class T> class tuple_element<I, volatile T>;
template<size_t I, class T> class tuple_element<I, const volatile T>;

Let TE denote tuple_element_t<I, T> of the cv-unqualified type T. Then specializations of each of the two templates meet the Cpp17TransformationTrait requirements with a member typedef type that names the following type:

(5.1) — for the first specialization, add_volatile_t<TE>, and
(5.2) — for the second specialization, add_cv_t<TE>.

In addition to being available via inclusion of the <tuple> (20.5.2) header, the two templates are available when any of the headers <array> (22.3.2), <ranges> (24.2), or <utility> (20.2.1) are included.

D.16 Variant

The header <variant> (20.7.2) has the following additions:

namespace std {
    template<class T> struct variant_size<volatile T>;
    template<class T> struct variant_size<const volatile T>;

    template<size_t I, class T> struct variant_alternative<I, volatile T>;
    template<size_t I, class T> struct variant_alternative<I, const volatile T>;
}

template<class T> class variant_size<volatile T>;
template<class T> class variant_size<const volatile T>;

Let VS denote variant_size<T> of the cv-unqualified type T. Then specializations of each of the two templates meet the Cpp17UnaryTypeTrait requirements with a base characteristic of integral_constant<size_t, VS::value>.

template<size_t I, class T> class variant_alternative<I, volatile T>;
template<size_t I, class T> class variant_alternative<I, const volatile T>;

Let VA denote variant_alternative<I, T> of the cv-unqualified type T. Then specializations of each of the two templates meet the Cpp17TransformationTrait requirements with a member typedef type that names the following type:

(3.1) — for the first specialization, add_volatile_t<VA::type>, and
(3.2) — for the second specialization, add_cv_t<VA::type>.

D.17 Deprecated iterator class template

The header <iterator> (23.2) has the following addition:
namespace std {
    template<class Category, class T, class Distance = ptrdiff_t,
        class Pointer = T*, class Reference = T&>
    struct iterator {
        using iterator_category = Category;
        using value_type = T;
        using difference_type = Distance;
        using pointer = Pointer;
        using reference = Reference;
    };
}

The `iterator` template may be used as a base class to ease the definition of required types for new iterators.

[Note 1: If the new iterator type is a class template, then these aliases will not be visible from within the iterator class's template definition, but only to callers of that class. —end note]

[Example 1: If a C++ program wants to define a bidirectional iterator for some data structure containing `double` and such that it works on a large memory model of the implementation, it can do so with:

```cpp
class MyIterator :
    public iterator<bidirectional_iterator_tag, double, long, T*, T&> {
        // code implementing ++, etc.
    };

--- end example]

D.18 Deprecated move_iterator access

[depr.move.iter.elem]

The following member is declared in addition to those members specified in 23.5.3.6:

```cpp
namespace std {
    template<class Iterator>
    class move_iterator {
        public:
            constexpr pointer operator->() const;
    };
}
```

Returns: current.

D.19 Deprecated shared_ptr atomic access

[depr.util.smartptr.shared.atomic]

The header `<memory>` (20.10.2) has the following additions:

```cpp
namespace std {
    template<class T>
    bool atomic_is_lock_free(const shared_ptr<T>* p);

    template<class T>
    shared_ptr<T> atomic_load(const shared_ptr<T>* p);
    template<class T>
    shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

    template<class T>
    void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
    template<class T>
    void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

    template<class T>
    shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
    template<class T>
    shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

    template<class T>
    bool atomic_compare_exchange_weak(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
    template<class T>
    bool atomic_compare_exchange_strong(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
```
```cpp
template<class T>
bool atomic_compare_exchange_weak_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);

template<class T>
bool atomic_compare_exchange_strong_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
}

Concurrent access to a shared_ptr object from multiple threads does not introduce a data race if the access is done exclusively via the functions in this subclause and the instance is passed as their first argument.

The meaning of the arguments of type memory_order is explained in 31.4.

```
template<class T>
bool atomic_compare_exchange_weak(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);

28 Requires: p shall not be null and v shall not be null.
29 Returns:
  atomic_compare_exchange_weak_explicit(p, v, w, memory_order::seq_cst, memory_order::seq_cst)
30 Throws: Nothing.

template<class T>
bool atomic_compare_exchange_strong(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);

31 Returns:
  atomic_compare_exchange_strong_explicit(p, v, w, memory_order::seq_cst, memory_order::seq_cst)

template<class T>
bool atomic_compare_exchange_weak_explicit(
  shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
  memory_order success, memory_order failure);

template<class T>
bool atomic_compare_exchange_strong_explicit(
  shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
  memory_order success, memory_order failure);

32 Requires: p shall not be null and v shall not be null. The failure argument shall not be
  memory_order::release nor memory_order::acq_rel.
33 Effects: If *p is equivalent to *v, assigns w to *p and has synchronization semantics corresponding to
  the value of success, otherwise assigns *p to *v and has synchronization semantics corresponding to
  the value of failure.
34 Returns: true if *p was equivalent to *v, false otherwise.
35 Throws: Nothing.
36 Remarks: Two shared_ptr objects are equivalent if they store the same pointer value and share
  ownership. The weak form may fail spuriously. See 31.8.2.

D.20 Deprecated basic_string capacity

1 The following member is declared in addition to those members specified in 21.3.3.5:

namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_string {
    public:
      void reserve();
  };
}

void reserve();

2 Effects: After this call, capacity() has an unspecified value greater than or equal to size().
  [Note 1: This is a non-binding shrink to fit request. — end note]

D.21 Deprecated standard code conversion facets

D.21.1 General

1 The header <codecvt> provides code conversion facets for various character encodings.

D.21.2 Header <codecvt> synopsis

namespace std {
  enum codecvt_mode {
    consume_header = 4,
    generate_header = 2,
    little_endian = 1
  };
}
D.21.3 Requirements

1 For each of the three code conversion facets `codecvt_utf8`, `codecvt_utf16`, and `codecvt_utf8_utf16`:
   - (1.1) `Elem` is the wide-character type, such as `wchar_t`, `char16_t`, or `char32_t`.
   - (1.2) `Maxcode` is the largest wide-character code that the facet will read or write without reporting a conversion error.
   - (1.3) If `MODE & consume_header`, the facet shall consume an initial header sequence, if present, when reading a multibyte sequence to determine the endianness of the subsequent multibyte sequence to be read.
   - (1.4) If `MODE & generate_header`, the facet shall generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written.
   - (1.5) If `MODE & little_endian`, the facet shall generate a multibyte sequence in little-endian order, as opposed to the default big-endian order.

2 For the facet `codecvt_utf8`:
   - (2.1) The facet shall convert between UTF-8 multibyte sequences and UCS-2 or UTF-32 (depending on the size of `Elem`) within the program.
   - (2.2) Endianness shall not affect how multibyte sequences are read or written.
   - (2.3) The multibyte sequences may be written as either a text or a binary file.

3 For the facet `codecvt_utf16`:
   - (3.1) The facet shall convert between UTF-16 multibyte sequences and UCS-2 or UTF-32 (depending on the size of `Elem`) within the program.
   - (3.2) Multibyte sequences shall be read or written according to the `MODE` flag, as set out above.
   - (3.3) The multibyte sequences may be written only as a binary file. Attempting to write to a text file produces undefined behavior.

4 For the facet `codecvt_utf8_utf16`:
   - (4.1) The facet shall convert between UTF-8 multibyte sequences and UTF-16 (one or two 16-bit codes) within the program.
   - (4.2) Endianness shall not affect how multibyte sequences are read or written.
   - (4.3) The multibyte sequences may be written as either a text or a binary file.
The encoding forms UTF-8, UTF-16, and UTF-32 are specified in ISO/IEC 10646. The encoding form UCS-2 is specified in ISO/IEC 10646:2003.\[332\]

**D.22 Deprecated convenience conversion interfaces** [depr.conversions]

**D.22.1 General** [depr.conversions.general]

The header `<locale>` (28.2) has the following additions:

```cpp
namespace std {
    template<class Codecvt, class Elem = wchar_t,
             class WideAlloc = allocator<Elem>,
             class ByteAlloc = allocator<char>>
    class wstring_convert;

    template<class Codecvt, class Elem = wchar_t,
             class Tr = char_traits<Elem>>
    class wbuffer_convert;
}
```

**D.22.2 Class template wstring_convert** [depr.conversions.string]

Class template `wstring_convert` performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template `codecvt`) to perform the conversions, without affecting any streams or locales.

*Example 1:* If you want to use the code conversion facet `codecvt_utf8` to output to `cout` a UTF-8 multibyte sequence corresponding to a wide string, but you don’t want to alter the locale for `cout`, you can write something like:

```cpp
wstring_convert<std::codecvt_utf8<wchar_t>> myconv;
std::string mbstring = myconv.to_bytes(L"Hello\n");
std::cout << mbstring;
```

--- end example]

```cpp
namespace std {
    template<class Codecvt, class Elem = wchar_t,
             class WideAlloc = allocator<Elem>,
             class ByteAlloc = allocator<char>>
    class wstring_convert {
        public:
            using byte_string = basic_string<char, char_traits<char>, ByteAlloc>;
            using wide_string = basic_string<Elem, char_traits<Elem>, WideAlloc>;
            using state_type = typename Codecvt::state_type;
            using int_type = typename wide_string::traits_type::int_type;

            wstring_convert() : wstring_convert(new Codecvt) {}
            explicit wstring_convert(Codecvt* pcvt);
            wstring_convert(Codecvt* pcvt, state_type state);
            explicit wstring_convert(const byte_string& byte_err,
                                      const wide_string& wide_err = wide_string());
            ~wstring_convert();

            wstring_convert(const wstring_convert&) = delete;
            wstring_convert& operator=(const wstring_convert&) = delete;

            wide_string from_bytes(char byte);
            wide_string from_bytes(const char* ptr);
            wide_string from_bytes(const byte_string& str);
            wide_string from_bytes(const char* first, const char* last);

            byte_string to_bytes(Elem wchar);
            byte_string to_bytes(const Elem* wptr);
            byte_string to_bytes(const wide_string& wstr);
            byte_string to_bytes(const Elem* first, const Elem* last);
    }
}
```

size_t converted() const noexcept;
state_type state() const;

private:
  byte_string byte_err_string; // exposition only
  wide_string wide_err_string; // exposition only
  Codecvt* cvtptr; // exposition only
  state_type cvtstate; // exposition only
  size_t cvtcount; // exposition only
};

The class template describes an object that controls conversions between wide string objects of class `basic_string<Elem, char_traits<Elem>, WideAlloc>` and byte string objects of class `basic_string<char, char_traits<char>, ByteAlloc>`. The class template defines the types `wide_string` and `byte_string` as synonyms for these two types. Conversion between a sequence of `Elem` values (stored in a `wide_string` object) and multibyte sequences (stored in a `byte_string` object) is performed by an object of class `Codecvt`, which meets the requirements of the standard code-conversion facet `codecvt<Elem, char, mbstate_t>`.

An object of this class template stores:

1. `byte_err_string` — a byte string to display on errors
2. `wide_err_string` — a wide string to display on errors
3. `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)
4. `cvtstate` — a conversion state object
5. `cvtcount` — a conversion count

```cpp
using byte_string = basic_string<char, char_traits<char>, ByteAlloc>;

size_t converted() const noexcept;

Returns: cvtcount.
```

```cpp
wide_string from_bytes(char byte);
wide_string from_bytes(const char* ptr);
wide_string from_bytes(const byte_string& str);
wide_string from_bytes(const char* first, const char* last);
```

**Effects:** The first member function shall convert the single-element sequence `byte` to a wide string. The second member function shall convert the null-terminated sequence beginning at `ptr` to a wide string. The third member function shall convert the sequence stored in `str` to a wide string. The fourth member function shall convert the sequence defined by the range `[first, last)` to a wide string.

In all cases:

1. If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.
2. The number of input elements successfully converted shall be stored in `cvtcount`.

**Returns:** If no conversion error occurs, the member function shall return the converted wide string. Otherwise, if the object was constructed with a wide-error string, the member function shall return the wide-error string. Otherwise, the member function throws an object of class `range_error`.

```cpp
using int_type = typename wide_string::traits_type::int_type;

The type shall be a synonym for `wide_string::traits_type::int_type`.
```

```cpp
state_type state() const;

Returns: cvtstate.
```

```cpp
using state_type = typename Codecvt::state_type;
```

**The type shall be a synonym for `Codecvt::state_type`.

§ D.22.2 1699
byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem* wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem* first, const Elem* last);

Effects: The first member function shall convert the single-element sequence wchar to a byte string. The second member function shall convert the null-terminated sequence beginning at wptr to a byte string. The third member function shall convert the sequence stored in wstr to a byte string. The fourth member function shall convert the sequence defined by the range [first, last) to a byte string.

In all cases:
— If the cvtstate object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.
— The number of input elements successfully converted shall be stored in cvtcount.

Returns: If no conversion error occurs, the member function shall return the converted byte string. Otherwise, if the object was constructed with a byte-error string, the member function shall return the byte-error string. Otherwise, the member function shall throw an object of class range_error.

using wide_string = basic_string<Elem, char_traits<Elem>, WideAlloc>;

The type shall be a synonym for basic_string<Elem, char_traits<Elem>, WideAlloc>.

explicit wstring_convert(Codecvt* pcvt);
wstring_convert(Codecvt* pcvt, state_type state);
explicit wstring_convert(const byte_string& byte_err,
const wide_string& wide_err = wide_string());

Requires: For the first and second constructors, pcvt != nullptr.

Effects: The first constructor shall store pcvt in cvtptr and default values in cvtstate, byte_err_string, and wide_err_string. The second constructor shall store pcvt in cvtptr, state in cvtstate, and default values in byte_err_string and wide_err_string; moreover the stored state shall be retained between calls to from_bytes and to_bytes. The third constructor shall store new Codecvt in cvtptr, state_type() in cvtstate, byte_err in byte_err_string, and wide_err in wide_err_string.

~wstring_convert();

Effects: The destructor shall delete cvtptr.

D.22.3 Class template wbuffer_convert

Class template wbuffer_convert looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template wstring_convert, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

namespace std {

template<class Codecvt, class Elem = wchar_t, class Tr = char_traits<Elem>>
class wbuffer_convert : public basic_streambuf<Elem, Tr> {
public:
    using state_type = typename Codecvt::state_type;

    wbuffer_convert() : wbuffer_convert(nullptr) {}
    explicit wbuffer_convert(streambuf* bytebuf, Codecvt* pcvt = new Codecvt,
                            state_type state = state_type());

    ~wbuffer_convert();

    wbuffer_convert(const wbuffer_convert&) = delete;
    wbuffer_convert& operator=(const wbuffer_convert&) = delete;

    streambuf* rdbuf() const;
    streambuf* rdbuf(streambuf* bytebuf);
}
state_type state() const;

private:
  streambuf* bufptr; // exposition only
  Codecvt* cvtptr; // exposition only
  state_type cvtstate; // exposition only
};

The class template describes a stream buffer that controls the transmission of elements of type Elem, whose character traits are described by the class Tr, to and from a byte stream buffer of type streambuf. Conversion between a sequence of Elem values and multibyte sequences is performed by an object of class Codecvt, which shall meet the requirements of the standard code-conversion facet codecvt<Elem, char, mbstate_t>.

An object of this class template stores:

(3.1) — bufptr — a pointer to its underlying byte stream buffer
(3.2) — cvtptr — a pointer to the allocated conversion object (which is freed when the wbuffer_convert object is destroyed)
(3.3) — cvtstate — a conversion state object

state_type state() const;

Returns: cvtstate.

streambuf* rdbuf() const;

Returns: bufptr.

streambuf* rdbuf(streambuf* bytebuf);

Effects: Stores bytebuf in bufptr.

Returns: The previous value of bufptr.

using state_type = typename Codecvt::state_type;

The type shall be a synonym for Codecvt::state_type.

explicit wbuffer_convert(
  streambuf* bytebuf,
  Codecvt* pcvt = new Codecvt,
  state_type state = state_type());

Requires: pcvt != nullptr.

Effects: The constructor constructs a stream buffer object, initializes bufptr to bytebuf, initializes cvtptr to pcvt, and initializes cvtstate to state.

-wbuffer_convert();

Effects: The destructor shall delete cvtptr.

D.23 Deprecated locale category facets  

The ctype locale category includes the following facets as if they were specified in table Table 102 of 28.3.1.2.1.

codecvt<char16_t, char, mbstate_t>
codecvt<char32_t, char, mbstate_t>

The ctype locale category includes the following facets as if they were specified in table Table 103 of 28.3.1.2.1.

codecvt_byname<char16_t, char, mbstate_t>
codecvt_byname<char32_t, char, mbstate_t>

The following class template specializations are required in addition to those specified in 28.4.2.5. The specialization codecvt<char16_t, char, mbstate_t> converts between the UTF-16 and UTF-8 encoding forms, and the specialization codecvt<char32_t, char, mbstate_t> converts between the UTF-32 and UTF-8 encoding forms.
D.24 Deprecated filesystem path factory functions  

-template<class Source>
   path u8path(const Source& source);
-template<class InputIterator>
   path u8path(InputIterator first, InputIterator last);

   Requires: The source and [first, last) sequences are UTF-8 encoded. The value type of Source and InputIterator is char or char8_t. Source meets the requirements specified in 29.11.6.4.

   Returns:

-(2.1) — If value_type is char and the current native narrow encoding (29.11.6.3.2) is UTF-8, return path(source) or path(first, last); otherwise,

-(2.2) — if value_type is wchar_t and the native wide encoding is UTF-16, or if value_type is char16_t or char32_t, convert source or [first, last) to a temporary, tmp, of type string_type and return path(tmp); otherwise,

-(2.3) — convert source or [first, last) to a temporary, tmp, of type u32string and return path(tmp).

Remarks: Argument format conversion (29.11.6.3.1) applies to the arguments for these functions. How Unicode encoding conversions are performed is unspecified.

Example 1: A string is to be read from a database that is encoded in UTF-8, and used to create a directory using the native encoding for filenames:

namespace fs = std::filesystem;
std::string utf8_string = read_utf8_data();
fs::create_directory(fs::u8path(utf8_string));

For POSIX-based operating systems with the native narrow encoding set to UTF-8, no encoding or type conversion occurs.
For POSIX-based operating systems with the native narrow encoding not set to UTF-8, a conversion to UTF-32 occurs, followed by a conversion to the current native narrow encoding. Some Unicode characters may have no native character set representation.
For Windows-based operating systems a conversion from UTF-8 to UTF-16 occurs.
—end example

Note 1: The example above is representative of a historical use of filesystem::u8path. To indicate a UTF-8 encoding, passing a std::u8string to path’s constructor is preferred as it is consistent with path’s handling of other encodings. —end note

D.25 Deprecated atomic operations  

D.25.1 General  

The header <atomic> (31.2) has the following additions.

namespace std {
   template<class T>
      void atomic_init(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
   template<class T>
      void atomic_init(atomic<T>*, typename atomic<T>::value_type) noexcept;

#define ATOMIC_VAR_INIT(value) see below
#define ATOMIC_FLAG_INIT see below
}

D.25.2 Volatile access  

If an atomic specialization has one of the following overloads, then that overload participates in overload resolution even if atomic<T>::is_always_lock_free is false:

void store(T desired, memory_order order = memory_order::seq_cst) volatile noexcept;
T operator=(T desired) volatile noexcept;
T load(memory_order order = memory_order::seq_cst) const volatile noexcept;
operator T() const volatile noexcept;
T exchange(T desired, memory_order order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_weak(T& expected, T desired,
      memory_order success, memory_order failure) volatile noexcept;
bool compare_exchange_strong(T& expected, T desired,
memory_order success, memory_order failure) volatile noexcept;
bool compare_exchange_weak(T& expected, T desired,
memory_order order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_strong(T& expected, T desired,
memory_order order = memory_order::seq_cst) volatile noexcept;
T fetch_key(T operand, memory_order order = memory_order::seq_cst) volatile noexcept;
T operator op=(T operand) volatile noexcept;
T* fetch_key(ptrdiff_t operand, memory_order order = memory_order::seq_cst) volatile noexcept;

D.25.3 Non-member functions

template<class T>
void atomic_init(volatile atomic<T>* object, typename atomic<T>::value_type desired) noexcept;

D.25.4 Operations on atomic types

#define ATOMIC_VAR_INIT(value) see below
1 The macro expands to a token sequence suitable for constant initialization of an atomic variable of
static storage duration of a type that is initialization-compatible with value.

[Note 1: This operation might need to initialize locks. — end note]
Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data
race.

[Example 1:]
atomic<int> v = ATOMIC_VAR_INIT(5);
— end example]

D.25.5 Flag type and operations

#define ATOMIC_FLAG_INIT see below
1 Remarks: The macro ATOMIC_FLAG_INIT is defined in such a way that it can be used to initialize an
object of type atomic_flag to the clear state. The macro can be used in the form:

atomic_flag guard = ATOMIC_FLAG_INIT;

It is unspecified whether the macro can be used in other initialization contexts. For a complete
static-duration object, that initialization shall be static.

§ D.25.5 1703
Bibliography

— ISO 4217:2015, Codes for the representation of currencies
— IANA Time Zone Database. Available from: https://www.iana.org/time-zones

The arithmetic specification described in ISO/IEC 10967-1:2012 is called LIA-1 in this document.
Cross references

Each clause and subclause label is listed below along with the corresponding clause or subclause number and page number, in alphabetical order by label.

accumulate (25.10.3) 1145
adjacent.difference (25.10.12) 1152
adjustfield.manip (29.5.6.2) 1394
alg.adjacent.find (25.6.8) 1092
alg.all.of (25.6.1) 1087
alg.any.of (25.6.2) 1087
alg.binary.search (25.8.4) 1120
alg.binary.search.general (25.8.4.1) 1120
alg.c.library (25.12) 1160
alg.clamp (25.8.10) 1138
alg.copy (25.7.1) 1099
alg.count (25.6.9) 1093
alg.equal (25.6.11) 1095
alg.find (25.7.6) 1106
alg.find.end (25.6.7) 1090
alg.find.first.of (25.6.8) 1091
alg.foreach (25.6.13) 1088
alg.generate (25.7.7) 1107
alg.heap.operations (25.8.8) 1132
alg.heap.operations.general (25.8.8.1) 1132
alg.is.permutation (25.6.12) 1096
alg.lex.comparison (25.6.13) 1139
alg.merge (25.6.8) 1125
alg.min.max (25.6.9) 1135
alg.modifying.operations (25.7) 1099
alg.move (25.7.2) 1101
alg.none.of (25.6.3) 1087
alg.nonmodifying (25.6) 1087
alg.nth.element (25.8.3) 1120
alg.partitions (25.8.5) 1123
alg.permutation.generators (25.8.13) 1140
alg.random.sample (25.7.12) 1113
alg.random.shuffle (25.7.13) 1114
alg.remove (25.7.8) 1107
alg.replace (25.7.5) 1104
alg.req (23.3.7) 947
alg.req.general (23.3.7.1) 947
alg.req.ind.cmp (23.3.7.5) 948
alg.req.ind.copy (23.3.7.3) 948
alg.req.ind.move (23.3.7.2) 947
alg.req.ind.swap (23.3.7.4) 948
alg.req.mergeable (23.3.7.7) 949
alg.req.permutable (23.3.7.6) 948
alg.req.sortable (23.3.7.8) 949
alg.reverse (25.7.10) 1111
alg.rotate (25.7.11) 1112
alg.search (25.6.13) 1097
alg.set.operations (25.8.7) 1127
alg.set.operations.general (25.8.7.1) 1127
alg.shift (25.7.14) 1114
alg.sort (25.8.2) 1115
alg.sorting (25.8) 1115
alg.sorting.general (25.8.1) 1115
alg.swap (25.7.3) 1102
alg.three.way (25.8.12) 1140
alg.transform (25.7.4) 1103
alg.unique (25.7.9) 1109
algorithm.stable (16.4.6.8) 503
algorithm.syn (25.4) 1049
algorithms (Clause 25) 1044
algorithms.general (25.1) 1044
algorithms.parallel (25.3) 1046
algorithms.parallel.defns (25.3.1) 1046
algorithms.parallel.exceptions (25.3.4) 1048
algorithms.parallel.exec (25.3.3) 1047
algorithms.parallel.overloads (25.3.5) 1049
algorithms.parallel.user (25.3.2) 1047
algorithms.requirements (25.2) 1044
algorithms.results (25.5) 1084
alloc.errors (17.6.4) 528
allocator.adaptor (20.13) 681
allocator.adaptor.cnstr (20.13.3) 683
allocator.adaptor.members (20.13.4) 684
allocator.adaptor.syn (20.13.1) 681
allocator.adaptor.types (20.13.2) 683
allocator.globals (20.10.10.3) 649
allocator.members (20.10.10.2) 648
allocator.requirements (16.4.4.6) 492
allocator.requirements.completeness (16.4.4.6.2) 497
allocator.requirements.general (16.4.4.6.1) 492
allocator.tag (20.10.7) 643
allocator.traits (20.10.9) 646
allocator.traits.general (20.10.9.1) 646
allocator.traits.members (20.10.9.3) 647
allocator.traits.types (20.10.9.2) 646
allocator.uses (20.10.8) 644
allocator.uses.construction (20.10.8.2) 644
allocator.uses.traits (20.10.8.1) 644
alt.headers (16.4.5.4) 500
any (20.8) 623
any.assign (20.8.4.3) 625
any.bad.any.cast (20.8.3) 623
any.class (20.8.4) 624
any.class.general (20.8.4.1) 624
any.cons (20.8.4.2) 624
any.general (20.8.1) 623
any.modifiers (20.8.4.4) 626
any.nonmembers (20.8.5) 627
fs.op.equivalent (29.11.13.13) 1495
fs.op.exists (29.11.13.14) 1496
fs.op.file.size (29.11.13.15) 1496
fs.op.funcs (29.11.13) 1490
fs.op.funcs.general (29.11.13.1) 1490
fs.op.hard.lk.ct (29.11.13.16) 1496
fs.op.is.block.file (29.11.13.17) 1496
fs.op.is.char.file (29.11.13.18) 1496
fs.op.is.directory (29.11.13.19) 1497
fs.op.is.empty (29.11.13.20) 1497
fs.op.is.fifo (29.11.13.21) 1497
fs.op.is.other (29.11.13.22) 1497
fs.op.is.regular.file (29.11.13.23) 1498
fs.op.is.socket (29.11.13.24) 1498
fs.op.is.symlink (29.11.13.25) 1498
fs.op.last.write.time (29.11.13.26) 1498
fs.op.permissions (29.11.13.27) 1499
fs.op.proximate (29.11.13.28) 1499
fs.op.read.symlink (29.11.13.29) 1499
fs.op.relative (29.11.13.30) 1499
fs.op.remove (29.11.13.31) 1500
fs.op.remove.all (29.11.13.32) 1500
fs.op.rename (29.11.13.33) 1500
fs.op.resize.file (29.11.13.34) 1500
fs.op.space (29.11.13.35) 1500
fs.op.status (29.11.13.36) 1500
fs.op.status.known (29.11.13.37) 1500
fs.op.symlink.status (29.11.13.38) 1500
fs.op.temp.dir.path (29.11.13.39) 1500
fs.op.weakly.canonical (29.11.13.40) 1500
fs.path.append (29.11.6.5.3) 1470
fs.path.assign (29.11.6.5.2) 1470
fs.path.compare (29.11.6.5.8) 1473
fs.path.concat (29.11.6.5.4) 1471
fs.path.construct (29.11.6.5.1) 1469
fs.path.cvt (29.11.6.3) 1467
fs.path.decompose (29.11.6.5.9) 1474
fs.path.fmt.cvt (29.11.6.3.1) 1467
fs.path.gen (29.11.6.5.11) 1475
fs.path.generic (29.11.6.2) 1466
fs.path.generic.obs (29.11.6.5.7) 1473
fs.path.io (29.11.6.7) 1477
fs.path.itr (29.11.6.6) 1476
fs.path.member (29.11.6.5) 1469
fs.path.modifiers (29.11.6.5.5) 1471
fs.path.native.obs (29.11.6.5.6) 1472
fs.path.nonmember (29.11.6.8) 1477
fs.path.query (29.11.6.5.10) 1475
fs.path.req (29.11.6.4) 1468
fs.path.type.cvt (29.11.6.3.2) 1468
fs.race.behavior (29.11.2.4) 1459
fs.req (29.11.6.5.3) 1459
fs.rec.dir.itr.members (29.11.12.2) 1489
fs.rec.dir.itr.nonmembers (29.11.12.3) 1490
fs.req (29.11.3) 1459
fs.req.general (29.11.3.1) 1459
fs.req.namespace (29.11.3.2) 1459
fstream (29.9.5) 1451
fstream.assign (29.9.5.3) 1452
fstream.cons (29.9.5.2) 1452
fstream.general (29.9.5.1) 1451
fstream.members (29.9.5.4) 1452
func.bind (20.14.15) 699
func.bind.bind (20.14.15.4) 700
func.bind.front (20.14.14) 699
func.bind.general (20.14.15.1) 699
func.bind.isbind (20.14.15.2) 699
func.bind.isplace (20.14.15.3) 700
func.bind.place (20.14.15.5) 701
func.def (20.14.3) 687
func.identity (20.14.12) 698
func.invoke (20.14.5) 688
func.memfn (20.14.16) 701
func.not.fn (20.14.13) 699
func.require (20.14.4) 687
func.search (20.14.18) 705
func.search.bm (20.14.18.3) 706
func.search.default (20.14.18.2) 705
func.search.general (20.14.18.1) 705
func.wrap (20.14.17) 701
func.wrap.badcall (20.14.17.2) 701
func.wrap.func (20.14.17.3) 702
func.wrap.func.alg (20.14.17.3.8) 705
func.wrap.func.cap (20.14.17.3.4) 704
func.wrap.func.con (20.14.17.3.2) 703
func.wrap.func.general (20.14.17.3.1) 702
func.wrap.func.inv (20.14.17.3.5) 704
func.wrap.func.mod (20.14.17.3.3) 704
func.wrap.func.nullptr (20.14.17.3.7) 704
func.wrap.func.targ (20.14.17.3.6) 704
func.wrap.general (20.14.17.1) 701
function.objects (20.14) 685
function.objects.general (20.14.1) 685
functional.syn (20.14.2) 685
functions.within.classes (16.3.3.4) 484
future.syn (32.9.2) 1616
futures (32.9) 1616
futures.async (32.9.3) 1617
futures.errors (32.9.3) 1617
futures.future.error (32.9.4) 1617
futures.overview (32.9.1) 1616
futures.promise (32.9.6) 1619
futures.shared.future (32.9.8) 1623
futures.state (32.9.5) 1618
futures.task (32.9.10) 1627
futures.task.general (32.9.10.1) 1627
futures.task.members (32.9.10.2) 1628
futures.task.nonmembers (32.9.10.3) 1629
futures.unique.future (32.9.7) 1621
get.new.handler (17.6.4.5) 529
get.terminate (17.9.5.3) 535
global.functions (16.4.6.4) 503
gram (Annex A) 1630
gram.basic (A.4) 1634
gram.class (A.9) 1646
gram.cpp (A.13) 1649
gram.dcl (A.7) 1639
gram.except (A.12) 1649
gram.expr (A.5) 1634
gram.general (A.1) 1630
gram.key (A.2) 1630
gram.lex (A.3) 1630
gram.module (A.8) 1645
gram.over (A.10) 1647
gram.stmt (A.6) 1638
gram.temp (A.11) 1647
gslice.access (26.7.6.3) 1225
gslice.array.assign (26.7.7.2) 1226
gslice.array.comp.assign (26.7.7.3) 1226
gslice.array.fill (26.7.7.4) 1226
gslice.cons (26.7.6.2) 1225
handler.functions (16.4.5.7) 501
hardware.interference (17.6.6) 529
hash.requirements (16.4.4.5) 491
headers (16.4.2.3) 485
hidden.friends (16.4.6.6) 503
ifstream (29.9.3) 1447
ifstream.assign (29.9.3.3) 1448
ifstream.cons (29.9.3.2) 1448
ifstream.general (29.9.3.1) 1447
ifstream.members (29.9.3.4) 1449
implimits (Annex B) 1651
includes (25.8.7.2) 1127
inclusive.scan (25.10.9) 1149
incrementable.traits (23.3.2.1) 930
indirect.array.assign (26.7.9.2) 1228
indirect.array.comp.assign (26.7.9.3) 1228
indirect.array.fill (26.7.9.4) 1228
indirectcallable (23.3.6) 946
indirectcallable.general (23.3.6.1) 946
indirectcallable.indirectinvocable (23.3.6.2) 946
initializer.list.syn (17.10.2) 537
inner.product (25.10.5) 1146
input.iterators (23.3.5.3) 959
input.output (Clause 29) 1375
input.output.general (29.1) 1375
input.streams (29.7.4) 1404
input.streams.general (29.7.4.1) 1404
insert.iter.ops (23.5.2.4.1) 959
insert.iterator (23.5.2.4) 959
insert.iterators (23.5.2) 957
insert.iterators.general (23.5.2.1) 957
inserter (23.5.2.4.2) 959
intro (Clause 4) 10
intro.abstract (4.1.2) 10
intro.compliance (4.1) 10
intro.compliance.general (4.1.1) 10
intro.defs (Clause 3) 3
intro.execution (6.9.1) 79
intro.memory (6.7.1) 58
intro.multithread (6.9.2) 81
intro.multithread.general (6.9.2.1) 81
intro.object (6.7.2) 59
intro.progress (6.9.2.3) 85
intro.races (6.9.2.2) 82
intro.refs (Clause 2) 2
intro.scope (Clause 1) 1
intro.structure (4.2) 11
intseq (20.3) 585
intseq.general (20.3.1) 585
intseq.intseq (20.3.2) 585
intseq.make (20.3.3) 585
invalid.argument (19.2.5) 567
ioanip.syn (29.7.3) 1403
ios (29.5.5) 1388
ios.base (29.5.3) 1381
ios.base.callback (29.5.3.7) 1387
ios.base.cons (29.5.3.8) 1387
ios.base.general (29.5.3.1) 1381
ios.base.locales (29.5.3.4) 1386
ios.base.storage (29.5.3.6) 1386
ios.failure (29.5.3.2.1) 1383
ios.fmtflags (29.5.3.2.2) 1383
ios.init (29.5.3.2.6) 1385
ios.iostate (29.5.3.2.3) 1383
ios.members.static (29.5.3.5) 1386
ios.openmode (29.5.3.2.4) 1383
ios.overview (29.5.5.1) 1388
ios.seekdir (29.5.3.2.5) 1383
ios.syn (29.5.1) 1379
ios.types (29.5.3.2) 1383
iosfwd.syn (29.3.1) 1376
isofstate.flags (29.5.5.4) 1392
iostream.assign (29.7.4.7.4) 1414
iostream.cons (29.7.4.7.2) 1414
iostream.dest (29.7.4.7.3) 1414
iostream.format (29.7) 1403
iostream.forward (29.3) 1376
iostream.forward.overview (29.3.2) 1378
iostream.limits.imbue (29.2.1) 1375
iostream.objects (29.4) 1378
iostream.objects.overview (29.4.2) 1378
iostream.syn (29.4.1) 1378
iostreamclass (29.7.4.7) 1414
iostreamclass.general (29.7.4.7.1) 1414
iostreams.base (29.5) 1379
iostreams.limits.pos (29.2.2) 1376
iostreams.requirements (29.2) 1375
iostreams.threadsafety (29.2.3) 1376
is_heap (25.8.8.6) 1134
is_sorted (25.8.2.5) 1119
istream (29.7.4.2) 1404
istream.assign (29.7.4.2.3) 1406
istream.cons (29.7.4.2.2) 1406
istream.extractors (29.7.4.3.3) 1408
istream.formatted (29.7.4.3) 1407
istream.formatted.arithmetic (29.7.4.3.2) 1407
istream.formatted.reqnts (29.7.4.3.1) 1407
istream.general (29.7.4.2.1) 1404
istream.iterator (23.6.2) 973
istream.iterator.cons (23.6.2.2) 974
istream.iterator.general (23.6.2.1) 973
Cross references 1715
move.iter.op.conv (23.5.3.5) 961
move.iter.requirements (23.5.3.3) 961
move.iterators (23.5.3.2) 960
move.iterators.general (23.5.3.1) 960
move.sent.ops (23.5.3.11) 964
move.sentinel (23.5.3.10) 963
multibyte.strings (16.3.3.3.5.3) 484
multimap (22.4.5) 875
multimap.cons (22.4.5.2) 878
multimap.erase (22.4.5.4) 879
multimap mod (22.4.5.3) 879
multimap.overview (22.4.5.1) 875
multiset (22.4.7) 882
multiset.cons (22.4.7.2) 885
multiset.erase (22.4.7.3) 885
multiset.overview (22.4.7.1) 882
mutex.syn (32.5.2) 1586
namespace.alias (9.8.3) 228
namespace.constraints (16.4.5.2) 497
namespace.def (9.8.2) 225
namespace.def.general (9.8.2.1) 225
namespace.future (16.4.5.2.3) 498
namespace.memdef (9.8.2.3) 227
namespace.qual (6.5.4.3) 49
namespace.std (16.4.5.2.1) 497
namespace.udecl (9.9) 231
namespace.udir (9.8.4) 229
namespace.unnamed (9.8.2.2) 226
narrow.stream.objects (29.4.3) 1379
new.badlength (17.6.4.2) 528
new.delete (17.6.3) 524
new.delete.array (17.6.3.3) 525
new.delete.dataareas (17.6.3.5) 528
new.delete.general (17.6.3.1) 524
new.delete.placement (17.6.3.4) 527
new.delete.single (17.6.3.2) 524
new.handler (17.6.4.3) 528
new.syn (17.6.2) 523
nullablepointer.requirements (16.4.4.4) 491
numarray (26.7) 1210
numbers (26.9) 1244
numbers.syn (26.9.1) 1244
numeric.iota (25.10.13) 1153
numeric.limits (17.3.5) 513
numeric.limits.general (17.3.5.1) 513
numeric.limits.members (17.3.5.2) 514
numeric.ops (25.10) 1145
numeric.ops.gcd (25.10.14) 1153
numeric.ops.general (25.10.1) 1145
numeric.ops.lcm (25.10.15) 1153
numeric.ops.midpoint (25.10.16) 1154
numeric.ops.overview (25.9) 1141
numeric.requirements (26.2) 1161
numeric.special (17.3.5.3) 518
numerics (Clause 26) 1161
numerics.defns (25.10.2) 1145
numerics.general (26.1) 1161
objects.within.classes (16.3.3.5) 485
ofstream (29.9.4) 1449
ofstream.assign (29.9.4.3) 1450
ofstream.cons (29.9.4.2) 1450
ofstream.general (29.9.4.1) 1449
ofstream.members (29.9.4.4) 1450
optional (20.6) 600
optional.assign (20.6.3.4) 604
optional.bad.access (20.6.5) 609
optional.comp.with.t (20.6.8) 610
optional ctor (20.6.3.2) 602
optional.dtor (20.6.3.3) 604
optional.general (20.6.1) 600
optional.hash (20.6.10) 612
optional.mod (20.6.3.7) 609
optional.nulls (20.6.7) 610
optional.nullopt (20.6.4) 609
optional.observe (20.6.3.6) 608
optional.optional (20.6.3) 601
optional.optional.general (20.6.3.1) 601
optional.relops (20.6.6) 609
optional.special (20.6.9) 611
optional.swap (20.6.3.5) 607
optional.syn (20.6.2) 600
organization (16.4.2) 485
organization.general (16.4.2.1) 485
ostream (29.7.5.2) 1415
ostream.assign (29.7.5.2.3) 1417
ostream.cons (29.7.5.2.2) 1417
ostream.formatted (29.7.5.3) 1418
ostream.formatted.reqmts (29.7.5.3.1) 1418
ostream.general (29.7.5.2.1) 1415
ostream.inserters (29.7.5.3.3) 1420
ostream.inserters.arithmetic (29.7.5.3.2) 1419
ostream.inserters.character (29.7.5.3.4) 1420
ostream.iterator (23.6.3) 975
ostream.iterator.cons.des (23.6.3.2) 975
ostream.iterator.general (23.6.3.1) 975
ostream.iterator.ops (23.6.3.3) 975
ostream.manip (29.7.5.5) 1422
ostream.rvalue (29.7.5.6) 1422
ostream.seeks (29.7.5.2.5) 1418
ostream.sentry (29.7.5.2.4) 1417
ostream.syn (29.7.2) 1403
ostream.unformatted (29.7.5.4) 1421
ostreambuf.constructor (23.6.5.2) 978
ostreambuf.iterator.ops (23.6.5.3) 978
ostreambuf.iterator (23.6.5) 977
ostreambuf.iterator.general (23.6.5.1) 977
ostreambuf.stream (29.8.4) 1436
ostreambuf.stream.assign (29.8.4.3) 1438
ostreambuf.stream.cons (29.8.4.2) 1437
ostreambuf.stream.general (29.8.4.1) 1436
ostreambuf.stream.members (29.8.4.4) 1438
out.of.range (19.2.7) 567
output.iterators (23.3.5.4) 943
output.streams (29.7.5) 1415

Cross references 1718
Cross references from ISO C++ 2020

All clause and subclause labels from ISO C++ 2020 (ISO/IEC 14882:2020, *Programming Languages — C++*) are present in this document, with the exceptions described below.

re.def see intro.refs
Index

Symbols

!, see operator, logical negation
!=, see operator, inequality
(), see operator, function call, see declarator, function
*, see operator, indirection, see operator, multiplication, see declarator, pointer
+, see operator, unary plus, see operator, addition
++, see operator, increment,
, see operator, comma
-, see operator, unary minus, see operator, subtraction
->, see operator, class member access
->*, see operator, pointer to member
--, see operator, decrement
.., see operator, class member access
.*, see operator, pointer to member
..., see ellipsis
/, see operator, division

block statement, 155
class declaration, 259
class definition, 259
definition, 221
initializer list, 202

_, see character, underscore
_++, 474
__cpp_aggregate_bases, 475
__cpp_aggregate_nsdecl, 475
__cpp_aggregate_paren_init, 475
__cpp_alias_templates, 475
__cpp aligned new, 475
__cpp_attributes, 475
__cpp_binary_literals, 475
__cpp_capture_star_this, 475
__cpp_char8_t, 475
__cpp_concepts, 475
__cpp conditional explicit, 475
__cpp constexpr, 475
__cpp constexpr init, 475
__cpp constexpr_init, 475
__cpp constexpr, 475
__cpp constexpr_dynamic_alloc, 475
__cpp constexpr_in_decltype, 475
__cpp constinit, 475
__cpp debase, 475
__cpp decltype, 475
__cpp decltype auto, 475
__cpp deduction guides, 475
__cpp delegating constructors, 475
__cpp designated_initializers, 475
__cpp enumerator attributes, 475
__cpp fold expressions, 475
__cpp generic lambdas, 475
__cpp guaranteed copy elision, 475
__cpp hex float, 475
__cpp_if constexpr, 475
__cpp_impl coroutine, 475
__cpp_impl destroying delete, 475
__cpp_impl_three way comparison, 475
__cpp inheriting constructors, 475
__cpp init captures, 475
__cpp initializer_lists, 475
__cpp inline variables, 475
__cpp lambda, 475
__cpp modules, 475
__cpp namespace attributes, 475
__cpp noexcept function type, 475
__cpp ntype template args, 475
__cpp ntype template parameter auto, 475

\, see operator, address-of, see operator, bitwise
AND, see declarator, reference
&&, see operator, logical AND
~, see operator, bitwise exclusive OR
\, see backslash character
{}
Index

class member, 119
default, 299
default argument, 300
friend function, 304
member function and, 269
member name, 299
multiple access, 308
nested class, 308
overload resolution and, 297
overloading and, 328
private, 299
protected, 299, 307
public, 299
using-declaration and, 236
virtual function, 307
access specifier, 301, 302
access-specifier, 289, 1647
accessible, 303
active
union member, 285
active macro directive, see macro, active
addition operator, see operator, addition
additive-expression, 139, 1637
address, 77, 142
addressable function, see function, addressable
aggregate, 202
elements, 202
aggregate deduction candidate, see candidate, aggregate deduction
aggregate initialization, 202
algorithm
stable, 8, 503
<algorithm>, 510, 512, 1049, 1667
alias
namespace, 228
alias template, see template, alias
alias-declaration, 164, 1640
alignas, 19, 240, 241, 1645
alignment, 68
extended, 69
fundamental, 69
new-extended, 69
stricter, 69
stronger, 69
weaker, 69
alignment requirement
implementation-defined, 68
alignment-specifier, 240, 1645
alignof, 19, 127, 130, 413, 414, 1637
allocated type, see type, allocated
allocation
alignment storage, 133
implementation-defined bit-field, 283
unspecified, 265
allocation functions, 66
alternate form
format string, 746
alternative token, see token, alternative ambiguity

A
abbreviated
template function, see template, function, abbreviated
abort, 90, 160
absolute path, see path, absolute
abstract class, see class, abstract
abstract-declarator, 185, 1642
abstract-pack-declarator, 185, 1643
access, 3
access control, 299–308
anonymous union, 287
base class, 302
base class member, 289

Numbers
0, see also zero, null
null character, see character, null
string terminator, 26

| operator, bitwise inclusive OR |
| operator, logical OR |
| operator, ones' complement, see destructor |

© ISO/IEC N4868
Index

base class member, 296
class conversion, 298
declaration type, 166
declaration versus cast, 185
declaration versus expression, 162
function declaration, 200
member access, 296
overloaded function, 328
parentheses and, 131
ambiguous conversion sequence, see conversion sequence, ambiguous
Amendment 1, 500
and, 19
and-expression, 143, 1638
and_eq, 19
anonymous union, 287
anonymous union object, 287
<any>, 510, 623, 1662
appearance-ordered, 88
appertain, 241
argv, 87
argument, 3, 501, 502, 568
  access checking and default, 300
  binding of default, 196
  evaluation of default, 196, 197
  example of default, 195, 196
  function call expression, 3
  function-like macro, 3
  overloaded operator and default, 354
  reference, 119
  scope of default, 197
  template, 369
  template instantiation, 3
  throw expression, 3
  type checking of default, 196
argument and name hiding
default, 197
argument and virtual function
default, 198
argument forwarding call wrapper, 688
argument list
  empty, 192
  variable, 192
argument passing, 119
  reference and, 207
argument substitution, see macro, argument substitution
argument type
  unknown, 192
argv, 87
arithmetic
  pointer, 139
unsigned, 75
array
  bound, 190
  const, 78
delete, 136
element, 190
  handler of type, 455
new, 132
overloading and pointer versus, 326
parameter of type, 192
sizeof, 130
template parameter of type, 364
$array$, 510–512, 598, 840, 843, 978, 1667, 1693
array
  as aggregate, 843
  contiguous storage, 843
  creation, 845
  initialization, 843, 844
tuple interface to, 845
  zero sized, 845
array size
default, 190
array type, 190
arrow operator, see operator, class member access
as-if rule, 11
asm, 19, 237, 1645
  implementation-defined, 237
asm-declaration, 237, 1645
assembler, 237
<assert.h>, 487, 569, 1679, 1683
assignment
  and lvalue, 146
  conversion by, 147
  copy, see assignment operator, copy
  move, 6, see assignment operator, move
  reference, 207
assignment operator
  copy, 269, 274–276
    hidden, 275
    implicitly declared, 274
    implicitly defined, 276
    non-trivial, 276
    trivial, 275
    virtual bases and, 276
  move, 269, 274–276
    hidden, 275
    implicitly declared, 275
    implicitly defined, 276
    non-trivial, 276
    trivial, 275
  overloaded, 355
assignment-expression, 147, 1638
assignment-operator, 147, 1638
associated, 1573
associated constraints, 377
associative containers
  exception safety, 828
  requirements, 828
  unordered, see unordered associative containers
asynchronous provider, 1618
asynchronous return object, 1618
at least as constrained, 379
at least as specialized as, see more specialized
atexit, 89
atomic
notifying operation, 1545
operation, 81–86
smart pointers, 1562–1567
waiting operation, 1545
eligible to be unblocked, 1546
atomic constraint, see constraint, atomic
attached
declaration, 249
entity, 56
attribute, 240–243
alignment, 241
carries dependency, 242
deprecated, 243
fallthrough, 243
likely, 244
maybe unused, 245
no unique address, 246
nodiscard, 245
noretur, 246
syntax and semantics, 240
unlikely, 244
attribute, 240, 1645
attribute-argument-clause, 240, 1645
attribute-declaration, 164, 1640
attribute-list, 240, 1645
attribute-namespace, 240, 1645
attribute-scoped-token, 240, 1645
attribute-specifier, 240, 1645
attribute-specifier-seq, 240, 1645
attribute-token, 240, 1645
attribute-using-prefix, 240, 1645
auto, 19, 159, 179, 1642
automatic storage duration, see storage duration, automatic
await-expression, 128, 1637
awk, 1512
B
backslash character, 22
bad_alloc, 134
bad_cast, 121
bad_typeid, 122
balanced-token, 240, 1645
balanced-token-seq, 240, 1645
barrier
phase synchronization point, 1615
<barrier>, 510, 1614, 1658
barrier phase, 1614
base characteristic, 708
base class, 288–290
dependent, 410
direct, 289
indirect, 289
non-virtual, 290
overloading and, 327
private, 302
protected, 302
public, 302
virtual, 290
base class subobject, 59
base prefix, 748
base-2 representation, 75
base-clause, 288, 1647
base-specifier, 289, 1647
base-specifier-list, 288, 1647
behavior
conditionally-supported, 4, 10
default, 4, 481
implementation-defined, 5, 11
locale-specific, 5
observable, 11
on receipt of signal, 81
required, 7, 481
undefined, 8, 10, 11, 976
unspecified, 8, 11
Ben, 328
Bernoulli distributions, 1195–1197
bernoulli_distribution
discrete probability function, 1195
Bessel functions
I_ν, 1240
J_ν, 1240
K_ν, 1240
N_ν, 1241
j_ν, 1243
n_ν, 1243
beta functions B, 1239
better conversion, see conversion, better
better conversion sequence, see conversion sequence, better
binary fold, 113
binary left fold, 113
binary operator
interpretation of, 355
overloaded, 355
binary operator function, see operator function, binary
binary right fold, 113
binary-digit, 20, 1632
binary-exponent-part, 23, 1633
binary-literal, 20, 1632
bind directly, 209
binding
reference, 207
binomial_distribution
discrete probability function, 1195
<bit>, 510, 511, 1170, 1658
bit-field, 283
address of, 284
alignment of, 284
implementation-defined alignment of, 283
implementation-defined sign of, 1677
type of, 283
unnamed, 283
zero width of, 284
bitand, 19
bitmask
   element, 483
   empty, 483
   value
      clear, 483
      is set, 483
      set, 483
bitor, 19
<biset>, 628
block (execution), 3, 1573, 1581, 1584, 1585, 1587,
   1588, 1590, 1591, 1605, 1609, 1611, 1612,
   1614, 1615, 1623, 1625
   with forward progress guarantee delegation, 86
block (statement), 3, see statement, compound
   initialization in, 162
   scope, 37
   structure, 162
block-declaration, 164, 1640
body
   function, 214
Bond
   James Bond, 110
bool, 19, 176, 1641
Boolean literal, 26
boolean literal, see literal, boolean
Boolean type, 75
boolean-literal, 26, 1634
bound argument entity, 687
bound arguments, 701
bound, of array, 190
brace-or-equal-initializer, 198, 1643
braced-init-list, 198, 1643
brains
   names that want to eat your, 498
break, 19, 160, 1639
buckets, 829
built-in candidate, 332
built-in operators, see operators, built-in
byte, 58, 130
C
C
   linkage to, 237
   standard, 1
   standard library, 2
c-char, 21, 1633
c-char-sequence, 21, 1633
C++ library headers
   importable, 485
call
   nodiscard, 245
   operator function, 354
call pattern, 688
call signature, 687
call wrapper, 687, 688
   forwarding, 688
   perfect forwarding, 688
   simple, 688
   type, 687
callable object, see object, callable
callable type, see type, callable, 702
candidate, 328
   aggregate deduction, 337
   usable, 329
capture
   implicit, 109
capture, 108, 1635
capture-default, 108, 1635
capture-list, 108, 1635
captured, 110
   by copy, 111
   by reference, 112
carries a dependency, 82
carry
   subtract_with_carry_engine, 1185
case, 19, 155, 157, 1639
<cassert>, 487, 569, 1679
cast
   base class, 124
   const, 126, 137
   derived class, 124
   dynamic, 121, 531
      construction and, 317
      destruction and, 317
   integer to pointer, 125
   lvalue, 123, 124
   pointer to integer, 125
   pointer-to-function, 125
   pointer-to-member, 124, 125
   reference, 123, 126
   reinterpret, 124, 137
      integer to pointer, 125
      lvalue, 124
      pointer to integer, 125
      pointer-to-function, 125
      pointer-to-member, 125
      reference, 126
   static, 123, 137
      lvalue, 123
      reference, 123
   undefined pointer-to-function, 125
cast-expression, 137, 1637
casting, 119
casting away constness, 126
catch, 19, 452
category tag, 949
cats
   interfering with canines, 529
caulky_distribution
      probability density function, 1203
<ccomplex>
   absence thereof, 499, 1658
<cctype>, 801, 1344
<cerrno>, 500, 569, 573
<cfenv>, 1161, 1162, 1667

Index
<cfloat>, 510, 520
char, 19, 176, 1641
  implementation-defined sign of, 75
char-like object, 760
char-like type, 760
char16_t, 19, see type, char16_t, 176, 1641
char32_t, 19, see type, char32_t, 176, 1641
char_class_type
  regular expression traits, 1507
class, 76, 259–285
  abstract, 295
    base, 500, 504
    cast to incomplete, 138
    constructor and abstract, 296
    definition, 31
    derived, 504
    implicit-lifetime, 261
    linkage of, 55
    linkage specification, 238
    local, see local class, 288
    member function, see member function, class	nested, 284
    polymorphic, 291
    scope of enumerator, 223
    standard-layout, 74, 260
    trivial, 74, 260
    trivially copyable, 74, 260
    union-like, 287
    unnamed, 170
  variant member of, 287
class, 19, 221, 259, 363, 1644, 1646, 1648
  class member access operator function, see
    operator function, class member access
class name
    elaborated, 176, 262
    point of declaration, 262
    scope of, 261
typedef, 170, 263
class object
    member, 265
    sizeof, 130
class object copy, see constructor, copy
class object initialization, see constructor
class-head, 259, 1646
class-head-name, 259, 1646
class-key, 259, 1646
class-name, 259, 1646
class-or-dectype, 289, 1647
class-specifier, 259, 1646
class-virt-specifier, 289, 1647
<climits>, 58, 510, 519, 1687
<clocale>, 484, 1374, 1679
  closure object, 104
  closure type, 104
<cmath>, 511, 1229, 1237, 1658, 1684
co_await, 19, 128
co_return, 19, 161
co_yield, 19, 145
<codecvt>, 1667, 1696
coherence
  read-read, 84
  read-write, 84
  write-read, 84
  write-write, 84
collating element, 4
closure operator, see operator, comma
comment, 15–17
  /* */, 16
  //, 16
combination
  separate, 13
compare-expression, 140, 1638
comparison
  pointer, 142
  pointer to function, 142
  undefined pointer, 140
comparison category types, 539
comparison operator function, see operator
  function, comparison
compatible with
  shared_ptr, 660
compiled
  separate, 13
compiler control line, see preprocessing directive
compl, 19
complete object, 59
complete object of, 60
complete-class context, 264
completely defined, 264
<complex>, 510, 1162, 1163, 1658, 1683, 1684
<complex.h>, 1678, 1683, 1684
component, 4
composite pointer type, 93
compound statement, see statement, compound
compound-requirement, 115, 1636
compound-statement, 155, 1639
concatenation
  macro argument, see ## operator
  string, 25
concept, 401
  model, 502
type, 402
concept, 19, 401, 1648
concept-definition, 401, 1648
concept-id, 369
concept-name, 401, 1648
<concepts>, 510, 554, 1658
concurrent forward progress guarantees, 85
condition, 154, 1639
conditions
  rules for, 154
<condition_variable>, 1604, 1667
conditional-expression
  throw-expression in, 144
conditional-expression, 144, 1638
conditionally-supported behavior, see behavior,
  conditionally-supported
conditionally-supported-directive, 461, 1650
conflict, 82
conformance requirements, 10–11
  class templates, 10
  classes, 10
  general, 10
  library, 10
  method of description, 10
conjunction, 375
consistency
  linkage, 167
  linkage specification, 239
  type declaration, 56
const, 19, 77, 184, 1642
  cast away, 126
  constructor and, 269, 270
  destructor and, 269, 277
  linkage of, 54
  overloading and, 326
  const member function, 268
  const object, see object, const
    undefined change to, 175
  const volatile member function, 268
  const volatile object, see object, const volatile
  const-default-constructible, 199
  const-qualified, 77
  const-volatile-qualified, 77
const_cast, 19, see cast, const, 117, 413, 414, 1636
const_local_iterator, 830
constant, 19, 100
  enumeration, 221
  null pointer, 98, 99
constant destruction, see destruction, constant
constant expression, 148, see expression, constant
  permitted result of, 152
constant initialization, 87
constant iterator, 929
constant subexpression, 4
constant-expression, 148, 1638
constant-initialized, 148
consteval, 19, 166, 1640
constexpr, 19, 155, 156, 166, 1639, 1640
constexpr function, 171
constexpr if, 156
constexpr iterators, 930
constexpr-compatible
  defaulted comparison operator, 321
  defaulted special member function, 269
constinit, 19, 166, 173, 1640, 1653
constituent expression, 79
constraint, 375
  associated, see associated constraints
  atomic, 376
  immediately-declared, 363
  normalization, 378–379
  satisfaction
    atomic, 377
    conjunction, 375
    disjunction, 375
  subsumption, 379
constraint-expression, 377, 1648
constraint-logical-and-expression, 361, 1648
constraint-logical-or-expression, 361, 1648
construction, 315–318
  dynamic cast and, 317
  member access, 315
  move, 6
  pointer to member or base, 316
  typeid operator, 317
  virtual function call, 317
constructor, 270
  address of, 270
  array of class objects and, 309
  converting, 280
  copy, 70, 269, 271–274, 485
  elision, 318
  implicitly declared, 272
  implicitly defined, 273
  non-trivial, 273
  trivial, 273
default, 269, 270
  non-trivial, 271
  trivial, 271
exception handling, see exception handling,
  constructors and destructors
explicit call, 270
implicitly called, 271
implicitly defined, 271
inheritance of, 270
inherited, 231
move, 269, 271–274
elision, 318
implicitly declared, 273
implicitly defined, 273
non-trivial, 273
trivial, 273
non-trivial, 270
random number distribution requirement, 1181
random number engine requirement, 1178
union, 286
constructor, conversion by, see conversion, user-defined
contained value
any, 624
optional, 602
variant, 614
container
contiguous, 810
reversible, 809
contains a value
optional, 602
context
non-deduced, 444
contextually converted constant expression of type bool, see conversion, contextual
contextually converted to bool, see conversion, contextual
contextually implicitly converted, 95
contiguous container, see container, contiguous
continue, 19, 160, 1639
and handler, 452
and try block, 452
control line, see preprocessing directive
control-line, 461, 1649
conventions, 481
lexical, 13–28
conversion
argument, 192
array-to-pointer, 96
better, 349
bool, 98
boolean, 99
class, 279
contextual, 95
contextual to bool, 95
contextual to constant expression of type bool, 151
deducted return type of user-defined, 282
derived-to-base, 344
floating to integral, 98
floating-point, 98
function pointer, 99
function-to-pointer, 96
implementation-defined pointer integer, 125
implicit, 95, 279
implicit user-defined, 279
inheritance of user-defined, 282
integer rank, 78
integral, 98
integral to floating, 98
lvalue-to-rvalue, 95, 1673
narrowing, 213
null member pointer, 99
null pointer, 98
overload resolution and, 340
overload resolution and pointer, 353
pointer, 98
pointer-to-member, 99
void*, 99
qualification, 97
return type, 161
standard, 95–99
temporary materialization, 96
to signed, 98
to unsigned, 98
type of, 281
user-defined, 279, 280
usual arithmetic, 99
virtual user-defined, 282
conversion explicit type, see casting
conversion function, see conversion, user-defined, 281
conversion rank, 345
conversion sequence
ambiguous, 344
better, 349
implicit, 343
indistinguishable, 349
standard, 95
user-defined, 345
worse, 349
coroutine, 217
promise type, 217
resumer, 218
<coroutine>, 511, 546, 547, 1658
coroutine return, see co_return

coroutine state, 218

coroutine-return-statement, 161, 1639

counted range, see range, counted

Cpp17Allocator, 492

Cpp17BidirectionalIterator, 945

Cpp17BinaryTypeTrait, 708

Cpp17Clock, 1259

Cpp17CopyAssignable, 489

Cpp17CopyConstructible, 489

Cpp17CopyInsertable into X, 811

Cpp17DefaultConstructible, 489

Cpp17DefaultInsertable into X, 810

Cpp17Destructible, 489

Cpp17EmplaceConstructible into X from args, 811

Cpp17EqualityComparable, 488

Cpp17Erasable from X, 811

Cpp17ForwardIterator, 944

Cpp17Hash, 491

Cpp17InputIterator, 942, 943

Cpp17Iterator, 942

Cpp17LessThanComparable, 488

Cpp17MoveAssignable, 489

Cpp17MoveConstructible, 489

Cpp17MoveInsertable into X, 810

Cpp17NullablePointer, 491

Cpp17OutputIterator, 943

Cpp17RandomAccessIterator, 945, 946

Cpp17TransformationTrait, 708

Cpp17UnaryTypeTrait, 708

<csignal>, 500, 550, 551, 1680

<ctalign> absence thereof, 499, 1658

<ctdarg>, 192, 500, 550

<ctdbool> absence thereof, 499, 1658

<ctddf> 130, 140, 507, 510, 1679, 1680

<ctdint> 75, 520, 1505, 1549, 1557, 1667

<ctdio> 522, 1378, 1379, 1385, 1444, 1503, 1504, 1679

<ctdlib> 89, 90, 487, 507, 508, 521, 522, 525, 550, 633, 649, 804, 1160, 1210, 1237, 1355, 1679, 1684

<cstring> 268, 484, 802, 1679, 1687, 1691

<ctgmath> absence thereof, 499, 1658

<ctime> 1334, 1336, 1679

ctor-initializer, 309, 1647

<type.h> 801, 1683

<char> 500, 804, 1667, 1678

ctor-initializer, 409

dependent member of the, 411

member of the, 410

currently handled exception, see exception handling, currently handled exception customization point, 498

currently handled exception, see exception handling, currently handled exception customization point, 498

cv-combined type, see type, cv-combined cv-decomposition, 97

cv-qualification signature, 97

cv-qualifier, 77

top-level, 78

cv-qualifier, 184, 1642

cv-qualifier-seq, 184, 1642

cv-unqualified type, see type, cv-unqualified

<cwchar>, 500, 763, 802–804, 1679

<cwctype>, 500, 801

D
d-char, 24, 1634

d-char-sequence, 24, 1634

DAG

multiple inheritance, 291

non-virtual base class, 291

virtual base class, 291

data member, see member, 264

non-static, 264

static, 264

data race, 84

deadlock, 4

deallocation function

usual, 67

deallocation functions, 66

decay

array, see conversion, array-to-pointer

function, see conversion, function-to-pointer
decimal-floating-point-literal, 23, 1633
decimal-literal, 20, 1632
dcl-reachable, 254
dcl-specifier, 166, 1640
dcl-specifier-seq, 166, 1640
declaration, 29, 164–243

array, 190

asm, 237

bit-field, 283

class name, 30

constant pointer, 187

default argument, 195–198

definition versus, 29

disqualifying, 502

eellips in function, 119, 192

enumerator point of, 36

exported, 249

extern, 29

extern reference, 207

forward, 167

forward class, 262

function, 29, 165, 192

local class, 288

member, 263

multiple, 56

name, 29

object, 165

opaque enum, 30

overloaded, 325

overloaded name and friend, 305

parameter, 30, 192
parentheses in, 185, 187
point of, 36–37
pointer, 187
reference, 188
**static** member, 30
storage class, 166
structured binding, see structured binding
declaration
type, 186
typedef, 165
typedef as type, 168
declaration, 164, 1640
declaration hiding, see name hiding
declaration-seq, 164, 1639
declaration-statement, 162, 1639
declarative region, 35
declarator, 30, 165, 183–214
array, 190
function, 191–195
meaning of, 186–198
multidimensional array, 191
pointer, 187
pointer-to-member, 189
reference, 188
declarator, 184, 1642
declarator-id, 184, 1642
declared specialization, see specialization, declared
dectype, 19, 178, 179, 1641, 1642
dectype-specifier, 178, 1641
decrement operator
overloaded, see overloading, decrement operator
decrement operator function, see operator function, decrement
deducible template, see template, deducible
deduction
class template argument, 383
class template arguments, 119, 176, 182, 336
placeholder type, 181
deduction-guide, 383, 1648
default, 19, 155, 214, 1639, 1643
default access control, see access control, default
default argument
overload resolution and, 340
default argument instantiation, 423
default constructor, see constructor, default
random number distribution requirement, 1181
seed sequence requirement, 1177
default initializers
overloading and, 327
default member initializer, 265
default memory resource pointer, 676
default-initialization, 199
default-inserted, 810
defaulted, 216
defered function, see function, deferred
define, 29
defined, 463
defined-macro-expression, 463, 1650
defining-type-id, 185, 1642
defining-type-specifier, 174, 1641
defining-type-specifier-seq, 174, 1641
definition, 29, 30
alternate, 500
class, 259, 263
class name as type, 261
constructor, 214
coroutine, 217
declaration as, 165
deleted, 216
function, 214–217
deleted, 216
explicitly-defaulted, 215
local class, 288
member function, 266
namespace, 225
nested class, 284
program semantics affected by, 422
pure virtual function, 295
scope of class, 261
static member, 283
virtual function, 294
definition domain, 33
definitions, 3–9
delete
array, 135
single-object, 135
delete, 19, 66, 135, 214, 323, 354, 413, 1637, 1643
destructor and, 136, 278
operator
replaceable, 500, 501
overloading and, 67
single-object, 135
type of, 323
undefined, 136
delete-expression, 135, 1637
deleted definition, see definition, deleted
deleted function, see function, deleted
deleter, 649
denormalized value, see number, subnormal
dependency-ordered before, 83
dependent base class, see base class, dependent
dependent member of the current instantiation, see current instantiation, dependent
member of the
dependent name, see name, dependent
<deque>, 510, 511, 840, 841, 978
dereferenceable iterator, see iterator, dereferenceable
dereferencing, see indirection
derivation, see inheritance
derived class, 288–296
most, see most derived class
derived object
most, see most derived object
designated-initializer-clause, 198, 1643
designated-initializer-list, 198, 1643
designator, 198, 1643
destroying operator delete, see operator delete, destroying
destruction, 315–318
  constant, 151
  dynamic cast and, 317
  member access, 315
  pointer to member or base, 316
typeid operator, 317
  virtual function call, 317
destructor, 276, 277, 485
default, 277
  exception handling, see exception handling,
    constructors and destructors
explicit call, 278
  implicit call, 277
  implicitly defined, 277
  non-trivial, 277
  program termination and, 277
  prospective, 276
  pure virtual, 277
  selected, 277
union, 286
  virtual, 277
diagnosable rules, 10
diagnostic message, see message, diagnostic
difference type, 929
digit, 18, 1631
digit-sequence, 23, 1633
digraph, see token, alternative, 16
direct base class, see base class, direct
direct member, see member, direct
direct-initialization, 200
direct-list-initialization, 209
direct-non-list-initialization, 4
directed acyclic graph, see DAG
directive, preprocessing, see preprocessing
directive-introducing token, see token,
  directive-introducing
directory, 1458
directory-separator, 1466
discard
  random number engine requirement, 1178
discard_block_engine
    generation algorithm, 1187
    state, 1186
    textual representation, 1187
    transition algorithm, 1187
discarded
  declaration, 254
discarded statement, 156
discarded-value expression, 94
discrete probability function
  bernoulli_distribution, 1195
  binomial_distribution, 1195
discrete_distribution, 1205
  discrete probability function, 1205
  weights, 1205
disjunction, 375
  disqualifying declaration, see declaration, disqualifying
disqualifying parameter, see parameter, disqualifying
distribution, see random number distribution
do, 19, 157, 1639
dogs
  obliviousness to interference, 529
domain error, 1238
dominance
  virtual base class, 298
dot
  filename, 1466
dot operator, see operator, class member access
dot-dot
  filename, 1466
double, 19, 24, 176, 1641
dynamic binding, see function, virtual
dynamic initialization, see initialization, dynamic
dynamic type, see type, dynamic
dynamic_cast, 19, see cast, dynamic, 117, 150, 413, 1636
E
  E (complete elliptic integrals), 1239
  E (incomplete elliptic integrals), 1241
  ECMA-262, 2
  ECMAScript, 1512, 1536
  egrep, 1512
  Ei (exponential integrals), 1242
  elaborated type specifier, see class name,
    elaborated
    elaborated_enum_specifier, 176, 1641
    elaborated_type_specifier, 176, 1641
  element access functions, 1046
  element type, 190
  elif-group, 461, 1650
  elif-groups, 461, 1650
eligible special member function, see special
  member function, eligible
eligible to be unblocked, 1546
eision
  copy, see constructor, copy, elision
  copy constructor, see constructor, copy,
    elision
  move constructor, see constructor, move,
    elision
eillation
  conversion sequence, 119, 346
  overload resolution and, 340
Index 1740
elliptic integrals
  complete Π, 1240
  complete E, 1239
  complete K, 1239
  incomplete Π, 1241
  incomplete E, 1241
  incomplete F, 1241
else, 19, 155, 156, 1639
else-group, 461, 1650
empty-declaration, 164, 1640
enclosing namespace set, 226
enclosing statement, 154
enclosing-name-space-specifier, 225, 1644
encoded character type, 1459
encoding
  multibyte, 26
encoding-prefix, 21, 1632
end-of-file, 633
endif-line, 461, 1650
engine, see random number engine
engine adaptor, see random number engine
adaptor
engines with predefined parameters
  default_random_engine, 1190
knuth_b, 1190
minstd_rand, 1189
minstd_rand0, 1189
mt19937, 1189
mt19937_64, 1189
ranlux24, 1190
ranlux24_base, 1190
ranlux48, 1190
ranlux48_base, 1190
entity, 29
  associated, 45
  implicitly movable, 319
  local, 29
  templated, 362
enum, 19, 76, 176, 221, 1641, 1644
  overloading and, 326
type of, 221, 222
  underlying type, see type, underlying
enum name
  typedef, 170
enum-base, 221, 1644
default_random_engine, 1190
knuth_b, 1190
std::minstd_rand, 1189
mt19937, 1189
mt19937_64, 1189
ranlux24, 1190
ranlux24_base, 1190
ranlux48, 1190
ranlux48_base, 1190
environment
  program, 87
equal, 1259
equality expression, see operator function, equality
equality_expression, 142, 1638
equivalence
  template type, 380
type, 169, 261
equivalent expressions, 396
  function templates, 397
  functionally, see functionally equivalent
template-heads, 397
template-parameters, 397
equivalent parameter declarations, 326
overloading and, 326
equivalent-key group, 828
equivalently-valued, 495
<errno.h>, 569, 1683
escape character, see backslash character
escape sequence
  format string, 743
  undefined, 22
escape-sequence, 22, 1633
Eulerian integral of the first kind, see beta
evaluation, 80
  order of argument, 118
  signal-safe, 551
  unspecified order of, 81, 88
  unspecified order of argument, 118
  unspecified order of function call, 118
exception
  arithmetic, 91
  undefined arithmetic, 91
<exception>, 512, 533
exception handling, 452–460
  constructors and destructors, 454
  currently handled exception, 456
exception object, 454
  constructor, 454
  destructor, 454
  function try block, 453
goto, 452
  handler, 452, 453, 455–457, 504
active, 456
array in, 455
incomplete type in, 455
match, 455–457
destructor call, 103
dynamic cast, 121
equality operators, 142
equality-preserving, 553
equivalent, see equivalent, expressions
equivalent, functionally equivalent, see functionally equivalent, expressions
increment, 120, 128
integral constant, 151
lambda, 104–113
left-shift-operator, 140
logical AND, 144
logical OR, 144
multiplicative operators, 139
new, 131
noexcept, 131
order of evaluation of, 91
parenthesized, 101
pointer-to-member, 138
pointer-to-member constant, 127
postfix, 116–127
potentially constant evaluated, 153
potentially evaluated, 31
primary, 100–116
pseudo-destructor call, 103
reference, 93
reinterpret cast, 124
relational operators, 141
requires, 113–116
right-shift-operator, 140
rvalue reference, 92
sizeof, 130
spaceship, 140
static cast, 123
three-way comparison, 140
throw, 146
type identification, 122
type-dependent, 408
unary, 127–131
unary operator, 127
value-dependent, 408
yield, 145
expression, 147, 1638
expression-equivalent, 5
expression-list, 117, 1636
expression-statement, 155, 1639
external, see namespace, extend
extended alignment, see alignment, extended
extended integer type, 75
extended signed integer type, 74
extended unsigned integer type, 74
extern, 19, 166, 237, 424, 1640, 1645, 1648
linkage of, 167
extern "C", 488, 500
extern "C++", 488, 500
extern template, see instantiation, explicit
external linkage, see linkage, external

**extreme_value_distribution**

probability density function, 1200

**F**

F (incomplete elliptic integrals), 1241

facet, 1339

fallback-separator, 1466

false, 19

*fenv.h*, 1162, 1683

file, 1458

file attributes, 1483
cached, 1483

file system, 1458

file system race, 1459

file, source, see source file

filename, 1466

filename, 1466

<filesystem>, 510, 511, 1459, 1662

final, 18, 259, 263, 292, 1646

final overrider, 292

final suspend point, 218

finite state machine, 5

**fisher_f_distribution**

probability density function, 1204

**float**

19, 24, 176, 1641

<float.h>, 520, 1683

floating-point literal, see literal, floating-point

floating-point promotion, 98

floating-point type, see type, floating-point
implementation-defined, 76

floating-point-literal, 23, 1633

floating-point-suffix, 24, 1633

fold

binary, 113

unary, 113

fold-expression, 113, 1636

fold-operator, 113, 1636

for, 19, 157, 159, 1639

scope of declaration in, 159

for-range-declaration, 157, 1639

for-range-initializer, 157, 1639

<format>, 511, 741, 1658

format specification

format string, 744

format specifier, 5

format string, 743

Formatter, 751

forward, 583

forward progress guarantees
concurrent, 85
delegation of, 86
parallel, 86
weakly parallel, 86

<forward_list>, 510, 511, 840, 841, 978, 1667

forwarding reference, 439

fractional-constant, 23, 1633

free store, see also delete, see also new, 323

freestanding implementation, see implementation, freestanding

friend

virtual and, 294
access speciﬁer and, 306
class access and, 304
inheritance and, 306
local class and, 306
template and, 389

friend, 19, 166, 176, 305, 1640

friend function
access and, 304
inline, 305
linkage of, 305
member function and, 304
nested class, 285

<fstream>, 1441, 1501

full-expression, 79

function, see also friend function, see also inline function, see also member function, see also virtual function

addressable, 497
allocation, 66, 132
conversion, 280
deallocation, 67, 323
deferred, 1626
definition, 31
deleted, 216
global, 500, 502
handler, 5
handler of type, 455
immediate, 171
inline, 173
linkage specification overloaded, 239
modifier, 6
named by expression or conversion, 32
needed for constant evaluation, 153
non-template, 194
observer, 6
operator, 353
template, 353
overload resolution and, 329
overloaded, see overloading
overloading and pointer versus, 326
parameter of type, 192
pointer to member, 139
program semantics affected by the existence of a function definition, 422
replacement, 7
reserved, 7
template parameter of type, 364
viable, 329
virtual, 291–295
pure, 295, 296
virtual function call, 117
virtual member, 500
waiting, 1618

function argument, see argument function call, 119
recursive, 119
undefined, 125
function call operator
overloaded, 355
function call operator function, see operator
function, function call
function object, 685
binders, 699–701
mem_fn, 701
reference_wrapper, 688
type, 685
wrapper, 701–705
function parameter, see parameter
function parameter pack, 386
function parameter scope, 38
function pointer type, 77
function return, see return
function return type, see return type
function try block, see exception handling,
function try block
function-body, 214, 1643
function-definition, 214, 1643
function-like macro, see macro, function-like
function-local predefined variable, see variable,
function-local predefined
function-specifier, 168, 1641
function-try-block, 452, 1649
<f functional>, 510–512, 685
functionally equivalent
expressions, 397
function templates, 397
template-heads, 397
functions
candidate, 415
fundamental alignment, see alignment,
fundamental
fundamental type, 76
destructor and, 279
fundamental type conversion, see conversion,
user-defined
future
shared state, 1618
<f future>, 1616, 1667

G
gamma_distribution
probability density function, 1199
generate
seed sequence requirement, 1177
generated destructor, see destructor, default
generation algorithm
discard_block_engine, 1187
independent_bits_engine, 1188
linear_congruential_engine, 1183
mersenne_twister_engine, 1184
shuffle_order_engine, 1188
subtract_with_carry_engine, 1185
generic lambda, 104
generic parameter type placeholder, 179
generic_distribution
discrete probability function, 1196
global module, see module, global
global module fragment, 253
global name, see name, global
global namespace, see namespace, global
global namespace scope, see namespace scope,
global
global scope, see scope, global
global-module-fragment, 253, 1646
glvalue, 92
goto, 19, 158, 160, 1639
and handler, 452
and try block, 452
initialization and, 162
generator, 1630
regular expression, 1536
grep, 1512
group, 461, 1649
group-part, 461, 1649

H
H_n (Hermite polynomials), 1242
h-char, 17, 1631
h-char-sequence, 17, 1631
h-pp-tokens, 463, 1650
h-preprocessing-token, 463, 1650
handler, see exception handling, handler
handler, 452, 1649
handler-seq, 452, 1649
happens after, 83
happens before, 83
hard link, 1458
has-attribute-expression, 463, 1650
has-include-expression, 463, 1650
hash
instantiation restrictions, 707
hash code, 829
hash function, 828
hash tables, see unordered associative containers
header, 485
C, 500, 502, 1684
C library, 488
C++ library, 485
importable, 252
name, 17
header unit, 252
preprocessing, 467
header-name, 17, 1631
header-name-tokens, 463, 1650
headers
C library, 1683
heap with respect to comp and proj, 1132
Hermite polynomials H_n, 1242
hex-quad, 14, 1630
hexadecimal-digit, 20, 1632
hexadecimal-digit-sequence, 20, 1632
hexadecimal-escape-sequence, 22, 1633
hexadecimal-floating-point-literal, 23, 1633
hexadecimal-fractional-constant, 23, 1633
hexadecimal-literal, 20, 1632
hexadecimal-prefix, 20, 1632
hiding, see name hiding
high-order bit, 58
hosted implementation, see implementation, hosted

I
Iν (Bessell functions), 1240
id
qualified, 103
id-expression, 101
id-expression, 101, 1635
identical
atomic constraints, see atomic constraint, identical
identifier, 17–18, 102, 165
identifier, 17, 1631
identifier label, 155
identifier-list, 462, 1650
identifier-nondigit, 17, 1631
if, 19, 155, 156, 158, 1639
if-group, 461, 1649
if-section, 461, 1649
ill-formed program, see program, ill-formed
immediate function, see function, immediate
immediate function context, 152
immediate invocation, 152
immediate subexpression, 79
implementation
freestanding, 10, 82, 86, 474, 487, 1543, 1545
hosted, 10, 474, 487
implementation limits, see limits, implementation
implementation-defined behavior, see behavior, implementation-defined
implementation-dependent, 426, 510, 1407, 1418
implementation-generated, 30
implicit conversion, see conversion, implicit
implicit conversion sequence, see conversion sequence, implicit
implicit object parameter, 329
implicit-lifetime class, see class, implicit-lifetime
implicit-lifetime type, see type, implicit-lifetime
implicitly movable entity, see entity, implicitly movable
implicitly-declared default constructor, see constructor, default, 270
implied object argument, 329
implicit conversion sequences, 329
non-static member function and, 329
import, 252
import, 18, 462, 467, 487, 1650
importable C++ library headers, see C++ library headers, importable
importable header, see header, importable
inclusion
conditional, see preprocessing directive, conditional inclusion
source file, see preprocessing directive, source-file inclusion
inclusive-or-expression, 143, 1638
incomplete, 139
incompletely-defined object type, see object type, incompletely-defined
increment operator
overloaded, see overloading, increment operator
increment operator function, see operator function, increment
incrementable, 937
independent_bits_engine
 generation algorithm, 1188
state, 1187
textual representation, 1188
transition algorithm, 1187
indeterminate value, see value, indeterminate, 64
indeterminately sequenced, 80
indirect base class, see base class, indirect
indirection, 127
inheritance, 288, 289
using-declaration and, 231
init-capture, 108, 1635
init-capture pack, 113, 386
init-declarator, 183, 1642
init-declarator-list, 183, 1642
init-statement, 154, 1639
initial suspend point, 218
initialization, 87, 198–214
aggregate, 202
array, 202
array of class objects, 206, 309
automatic, 162
base class, 309, 310
by inherited constructor, 314
class array, 206, 207
class member, 200
class object, see also constructor, 202, 308–315
cost, 174, 202
cost member, 311
costant, 87
constructor and, 308, 309
copy, 200
default, 199
default constructor and, 308
definition and, 165
direct, 200
dynamic, 87
dynamic block-scope, 162
dynamic non-local, 88
explicit, 309
jump past, 162
list-initialization, 209–214
local static, 162

Index 1745
Index

inter-thread happens before, 83
interface dependency, 253
internal linkage, see linkage, internal
interval boundaries
  piecewise_constant_distribution, 1207
  piecewise_linear_distribution, 1208
<inttypes.h>, 1505, 1683
invalid iterator, see iterator, invalid
invalid pointer value, see value, invalid pointer
invocation
  macro, 469
<iomanip>, 512, 1403, 1423, 1424
<iomanip>, 512, 1403, 1423, 1424
<iostream>, 1379
<iostream>, 5, 1376, 1377
<iostream>, 405, 1378, 1385
isctype
  regular expression traits, 1507
<intos64.h>, 486, 1679, 1683, 1684
<istream>, 510, 1403, 1404
iteration-statement, 157, 160, 1639
iterator, 929
  constexpr, 930
  dereferenceable, 929
  invalid, 930
  past-the-end, 929
<iterator>, 510–512, 922, 978, 986, 1693
J
j_n (spherical Bessel functions), 1243
J_n (Bessell functions), 1240
Jessie, 280
jump-statement, 160, 1639
K
K (complete elliptic integrals), 1239
K_n (Bessell functions), 1240
key parameter, see parameter, key
keyword, 19, 1630
keyword, 18, 1631
L
L_n (Laguerre polynomials), 1242
L_n^m (associated Laguerre polynomials), 1238
label, 161
  case, 155, 157
  default, 155, 157
  scope of, 38, 155
labeled-statement, 155, 1639
Laguerre polynomials
  L_n, 1242
  L_n^m, 1238
lambda-capture, 108, 1635
lambda-declarator, 104, 1635
lambda-expression, 104, 1635
lambda-introducer, 104, 176, 1635
language linkage, 237
<latch>, 511, 1613, 1658
lattice, see DAG, see subobject
layout
- bit-field, 283
- class object, 265, 290
layout-compatible, 74
  - class, 266
  - enumeration, 223
layout-compatible type, 74
left shift
  - undefined, 140
left shift operator, see operator, left shift
left-pad, 755
Legendre functions $Y_m^\ell$, 1243
Legendre polynomials
  - $P_\ell$, 1242
  - $P_m^\ell$, 1239
letter, 484
lexical conventions, see conventions, lexical
LIA-1, 1704
library
- C standard, 479, 483, 485, 488, 1678, 1683
- C++ standard, 478, 500, 501, 504
library clauses, 11
lifetime, 61
limits
  - implementation, 5
  - `<limits>`, 510, 512
  - `<limits.h>`, 519, 1683
line number, 473
line splicing, 13
linear_congruential_engine
  - generation algorithm, 1183
  - modulus, 1183
  - state, 1183
  - textual representation, 1183
  - transition algorithm, 1183
link, 1458
linkage, 29, 54–56
  - `const` and, 54
  - external, 54, 488, 500
  - implementation-defined object, 240
  - `inline` and, 54
  - internal, 54
  - module, 54
  - no, 54, 56
static and, 54
linkage specification, see specification, linkage
  - `linkage-specification`, 237, 1645
  - `<list>`, 510, 511, 840, 842, 978
list-initialization, 209
literal, 19–28, 100
  - base of integer, 20
  - boolean, 26
  - `char16_t`, 22
  - `char32_t`, 22
  - character, 21, 22
    - ordinary, 22
    - UTF-16, 22
    - UTF-32, 22
    - UTF-8, 22
  - complex, 1170
  - constant, 19
  - `float`, 24
  - floating-point, 23, 24
  - implementation-defined value of `char`, 23
  - integer, 20, 21
  - `long`, 20, 21
  - `long double`, 24
  - multicharacter, 22
    - implementation-defined value of, 22
  - narrow-character, 25
  - operator, 359
    - raw, 360
    - template, 359
    - template numeric, 360
    - template string, 360
  - pointer, 26
  - string, 24, 25
    - `char16_t`, 25
    - `char32_t`, 25
    - narrow, 25
    - raw, 15, 24
    - undefined change to, 26
    - UTF-16, 25
    - UTF-32, 25
    - UTF-8, 25
    - wide, 25
  - suffix identifier, 359
  - type of character, 22
  - type of floating-point, 24
  - type of integer, 21
  - `unsigned`, 20, 21
  - user-defined, 26
  - `literal`, 20, 1632
  - literal type, see type, literal
  - `literal-operator-id`, 359, 1647
  - living dead
    - name of, 498
  - local class, see class, local
  - friend, 306
    - member function in, 267
  - scope of, 288
  - local entity, see entity, local
  - local scope, see scope, block
  - local variable, see variable, local
    - destruction of, 160, 162
  - `local_iterator`, 830
  - locale, 1506, 1508, 1512
  - `<locale>`, 510, 1335, 1336, 1698
  - locale-specific behavior, see behavior, locale-specific
  - locale-specific form
    - format string, 747
  - `<locale.h>`, 1374, 1683
  - lock-free execution, 85
  - `logical-and-expression`, 144, 1638
  - `logical-or-expression`, 144, 1638
  - lognormal_distribution

Index 1747
probability density function, 1202
long, 19, 24, 176, 1641
typedef and, 166
long-long-suffix, 20, 1632
long-suffix, 20, 1632
lookup
argument-dependent, 45
class member, 48, 53
elevated type specifier, 52–53
member name, 296
name, 29, 41–54
namespace aliases and, 54
namespace member, 49
qualified name, 47–52
template name, 402
unqualified name, 41
using-directives and, 54
lookup_classname
regular expression traits, 1507, 1538
lookup_collatename
regular expression traits, 1507
low-order bit, 58
lowercase, 484
lparen, 461, 1650
lvalue, 92, 1673
lvalue reference, 188
Lvalue-Callable, 702

M
macro
active, 467
argument substitution, 469
definition, 467
function-like, 468, 469
arguments, 469
import, 467–468
masking, 502
name, 468
object-like, 468, 469
point of definition, 467
point of import, 467
point of undeinition, 467
pragma operator, 476
predefined, 474
replacement, 468–473
replacement list, 468
rescanning and replacement, 472
scope of definition, 473
main function, 86–87
implementation-defined linkage of, 87
implementation-defined parameters to, 87
parameters to, 87
return from, 87, 89
make progress
thread, 85
make-unsigned-like-t, 985
manifestly constant-evaluated, 152
<map>, 510, 511, 868, 978

match_results
as sequence, 1523
matched, 5
<math.h>, 1237, 1683
mathematical special functions, 1238–1243
max
random number distribution requirement, 1181
mean
normal_distribution, 1201
poisson_distribution, 1198
mem-initializer, 310, 1647
mem-initializer-id, 310, 1647
mem-initializer-list, 310, 1647
member
class static, 65
default initializer, 265
direct, 263
enumerator, 223
non-static, 264
static, 264, 282
template and static, 383
member access operator
overloaded, 356
member candidate, 332
member data
static, 283
member function, 264
call undefined, 267
class, 266
const, 268
const volatile, 268
constexpr-compatible, 269
constructor and, 270
destructor and, 277
friend, 305
inline, 266
local class, 288
nested class, 308
non-static, 264, 267
overload resolution and, 329
static, 264, 282
this, 268
union, 286
volatile, 268
member names, 38
member of an unknown specialization, 411
member of the current instantiation, see current instantiation, member of the
member pointer to, see pointer to member
member subobject, 59
member-declaration, 263, 1646
member-declarator, 263, 1646
member-declarator-list, 263, 1646
member-specification, 263, 1646
members, 38
<memory>, 510–512, 633, 634, 1154, 1694
memory location, 58
memory management, see delete, see new
Index
exception specification, 459
needed for constant evaluation, 153
**negative_binomial_distribution**
discrete probability function, 1197
nested class, see class, nested
  local class, 288
  scope of, 284
nested within, 60
**nested-name-specifier**, 103, 1635
**nested-namespace-definition**, 225, 1644
**nested-requirement**, 116, 1636
Neumann functions
  \( N_\nu \), 1241
  \( n_n \), 1243
**<new>**, 66, 511, 523
**new**, 19, 66, 131, 132, 354, 413, 1637
  array of class objects and, 134
  constructor and, 134
  default constructor and, 134
  exception and, 135
  initialization and, 134
  **operator**
  replaceable, 500, 501
  scoping and, 132
  storage allocation, 131
  type of, 323
  unspecified constructor and, 134
  unspecified order of evaluation, 134
**new-declarator**, 131, 1637
**new-expression**, 131, 1637
  placement, 134
**new-extended alignment**, see **alignment**, new-extended
**new-initializer**, 131, 1637
**new-line**, 462, 1650
**new-placement**, 131, 1637
**new-type-id**, 131, 1637
**new_handler**, 67
  no linkage, 54
  node handle, 816
  **nodeclspec-function-declaration**, 164, 1640
  nodiscard call, see call, nodiscard
  nodiscard type, see type, nodiscard
**noexcept**, 19, 115, 131, 413, 414, 1636, 1637
**noexcept-expression**, 131, 1637
**noexcept-specifier**, 457, 1649
  non-initialization odr-use, see odr-use,
  non-initialization
  non-member candidate, 332
  non-static data member, see data member,
  non-static
  non-static member, see member, non-static
  non-static member function, see member function,
  non-static
  non-template function, see function, non-template
  non-throwing exception specification, 457
  non-virtual base class, see base class, non-virtual
**nondigit**, 18, 1631
  **nonzero-digit**, 20, 1632
**noptr-abstract-declarator**, 185, 1642
**noptr-abstract-pack-declarator**, 185, 1643
**noptr-declarator**, 184, 1642
**noptr-new-declarator**, 131, 1637
normal distributions, 1201–1205
  normal form
    constraint, 378
    path, 1467
**normal_distribution**
  mean, 1201
  probability density function, 1201
  standard deviation, 1201
normalization
  constraint, see constraint, normalization
  path, see path, normalization
normative references, see references, normative
**not**, 19
**not_eq**, 19
notation
  **syntax**, 12
NTBS, 484, 1690, 1691
  empty, 484
  length, 484
  static, 484
  value, 484
NTCTS, 6
**ntmbs**, 484
  static, 484
null character, see character, null
null member pointer conversion, see conversion, null member pointer
null pointer conversion, see conversion, null pointer
null pointer value, see value, null pointer
null statement, 155
null wide character, see wide-character, null
**nullptr**, 19
number
  **hex**, 22
  **octal**, 22
  preprocessing, 17
  subnormal, 513, 514, 516, 517
**<numbers>**, 511, 1244, 1658
**<numeric>**, 511, 512, 1141
numeric type, see type, numeric
**numeric_limits**, 512
  specializations for arithmetic types, 76
O
object, see also object model, 29, 59
  byte copying and, 72–73
  callable, 687
  complete, 59
  cost, 78
  const volatile, 78
  definition, 31
  destructor and placement of, 278
  destructor static, 89
exception, see exception handling, exception
object
implicit creation, 60
linkage specification, 240
local static, 65
nested within, 60
nonzero size, 60
providing storage for, 59
reified, 986
suitable created, 61
unnamed, 270
volatile, 78
zero size, 60
object class, see class object
object expression, 120, 138
object lifetime, 61–64
object model, 59–61
object pointer type, 77
object temporary, see temporary
object type, 74
incompletely-defined, 73
object-like macro, see macro, object-like
observable behavior, see behavior, observable
octal-digit, 20, 1632
octal-escape-sequence, 22, 1633
octal-literal, 20, 1632
odr-usable, 32
odr-use, 32
non-initialization, 88
one-definition rule, 31–35
opaque-enum-declaration, 221, 1644
operating system dependent, 1458
operator, 19, 354
* =, 146
+=, 128, 146
-=, 146
/=, 146
<< =, 146
>> =, 146
%=, 146
& =, 146
| =, 146
l =, 146
addition, 139
additive, 139
address-of, 127
assignment, 146, 485
bitwise, 143
bitwise AND, 143
bitwise exclusive OR, 143
bitwise inclusive OR, 143
cast, 127, 184
class member access, 119
comma, 147
comparison
constexpr-compatible, 321
implicitly defined, 320
secondary, 322
conditional expression, 144
copy assignment, see assignment operator, copy
decrement, 121, 127, 128
division, 139
equality, 142
defaulted, 321
deleted, 320
function call, 117, 354
greater than, 141
greater than or equal to, 141
implementation, 353
increment, 120, 127, 128
indirection, 127
inequality, 142
defaulted, 322
left shift, 140
less than, 141
less than or equal to, 141
logical AND, 144
logical negation, 127, 128
logical OR, 144
move assignment, see assignment operator, move
multiplication, 139
multiplicative, 139
ones’ complement, 127, 128
overloaded, 91, 353
pointer to member, 138
pragma, see macro, pragma operator
precedence of, 91
relational, 141
defaulted, 322
remainder, 139
right shift, 140
scope resolution, 48, 103, 132, 266, 289, 295
side effects and comma, 147
side effects and logical AND, 144
side effects and logical OR, 144
sizeof, 127, 130
spaceship, 140
subscripting, 117, 354
subtraction, 139
three-way comparison, 140
defaulted, 322
deleted, 320
unary, 127
unary minus, 127, 128
unary plus, 127, 128
operator, 353, 1647
operator, 19, 280, 331, 333–335, 341, 344, 350,
353–357, 359, 1647
operator delete
destroying, 67
operator delete, see also delete, 132, 136, 323
operator function
binary, 355
class member access, 356
comparison, 355
decrement, 357
equality, 355
function call, 355
increment, 356
prefix unary, 354
relational, 355
simple assignment, 355
subscripting, 356
three-way comparison, 355

operator new, see also new, 132
operator overloading, see overloading, operator
operator use
  scope resolution, 283
operator!=
  random number distribution requirement, 1181
  random number engine requirement, 1179
operator()
  random number distribution requirement, 1181
  random number engine requirement, 1178
operator-function-id, 353, 1647
operator-or-punctuator, 19, 1631
operator<<
  random number distribution requirement, 1182
  random number engine requirement, 1179
operator==
  random number distribution requirement, 1181
  random number engine requirement, 1179
operator>>
  random number distribution requirement, 1182
  random number engine requirement, 1179
operators
  built-in, 91
optimization of temporary, see temporary, elimination of
<optional>, 511, 600, 1662
optional object, 600
or, 19
or_eq, 19
order of evaluation in expression, see expression, order of evaluation of
order of execution
  base class constructor, 271
  base class destructor, 277
  constructor and array, 308
  constructor and static data members, 309
  destructor, 277
  destructor and array, 277
  member constructor, 271
  member destructor, 277
ordering
  function template partial, see template, function, partial ordering
ordinary character literal, 22
ordinary string literal, 25
<ostream>, 510, 1403, 1415
over-aligned type, see type, over-aligned
overflow, 91
undefined, 91
overload resolution, 325
overload set, 41
overloaded function, see overloading
  address of, 128, 352
overloaded operator, see overloading, operator
  inheritance of, 354
overloading, 193, 261, 325–359, 395
  access control and, 328
  address of overloaded function, 352
  argument lists, 329–340
  array versus pointer, 326
  assignment operator, 355
  binary operator, 355
  built-in operators and, 357
  candidate functions, 329–340
  declaration matching, 327
  declarations, 325
  example of, 325
  function call operator, 355
  function versus pointer, 326
  member access operator, 356
  operator, 353–357
  prohibited, 325
  resolution, 328–352
    best viable function, 340–354
    better viable function, 341
    contexts, 328
    function call syntax, 330–332
    function template, 450
    implicit conversions and, 343–352
    initialization, 335, 336
    operators, 332
    scoping ambiguity, 297
    template, 397
    template name, 402
    viable functions, 340–354
    subscripting operator, 356
    unary operator, 354
    user-defined literal, 359
    using directive and, 230
    using-declaration and, 236
overloads
  floating-point, 1170
override, 18, 263, 293, 1646
overrider
  final, 292
own, 649

P
P_ℓ (Legendre polynomials), 1242
P^m_ℓ (associated Legendre polynomials), 1239
pack, 386
  unexpanded, 387
pack expansion, 386
  pattern, 386

Index
padding bits, 73
pair
tuple interface to, 586
parallel algorithm, 1046
parallel forward progress guarantees, 86
param
random number distribution requirement, 1181
seed sequence requirement, 1177
param_type
random number distribution requirement, 1181
parameter, 6
catch clause, 6
disqualifying, 561
function, 6
function-like macro, 6
key, 562
macro, 469
reference, 188
scope of, 38
template, 6, 30
void, 192
parameter declaration, 30
parameter list
variable, 119, 192
parameter mapping, 376
parameter-declaration, 192, 1643
parameter-declaration-clause, 192, 1643
parameter-declaration-list, 192, 1643
parameter-type-list, 192
parameterized type, see template
parameters-and-qualifiers, 184, 1642
parent directory, 1458
past-the-end iterator, see iterator, past-the-end
path, 1463
absolute, 1463
normalization, 1466–1467
relative, 1463
path equality, 1477
pathname, 1463
pathname resolution, 1463
pattern, see pack expansion, pattern
perfect forwarding call wrapper, 688
period, 484
phase completion step, 1615
phase synchronization point, see barrier, phase
synchronization point
phases of translation, see translation, phases
Π (complete elliptic integrals), 1240
Π (incomplete elliptic integrals), 1241
piecewise construction, 588
piecewise_constant_distribution
interval boundaries, 1207
probability density function, 1208
weights at boundaries, 1208
placeholder type deduction, 181
placeholder-type-specifier, 179, 1642
placement new-expression, see new-expression, placement
plain lock-free atomic operation, 551
pm-expression, 138, 1637
POD, 1692
point, 77
point of declaration, see declaration, point of
macro definition, see macro, point of definition
macro import, see macro, point of import
macro undefined, see macro, point of undefined
pointer, see also void*
composite pointer type, 93
integer representation of safely-derived, 68
safely-derived, 68
strict total order, 5
to traceable object, 68, 505
zero, see value, null pointer
pointer literal, see literal, pointer
pointer past the end of, 77
pointer to, 77
pointer to member, 76, 138, 189
pointer-interconvertible, 77
pointer-literal, 26, 1634
Poisson distributions, 1197–1201
poisson_distribution
discrete probability function, 1197
mean, 1198
polymorphic class, see class, polymorphic
pool resource classes, 677
pools, 677
population, 1113
POSIX, 2
extended regular expressions, 1512
regular expressions, 1512
postfix ++ and --
overloading, 356
postfix ++, 120
postfix --, 121
postfix-expression, 117, 1636
potential results, 31
potential scope, 35
potentially concurrent, 84
potentially constant evaluated, 153
potentially evaluated, 31
potentially-constant, 148
potentially-overlapping subobject, 60
potentially-throwing
expression, 458
pp-global-module-fragment, 461, 1649
pp-import, 467, 1650
pp-module, 466, 1650
pp-number, 17, 1631
pp-private-module-fragment, 461, 1649
pp-tokens, 462, 1650
precedence of operator, see operator, precedence of
preferred-separator, 1466
prefix
L, 22, 25
R, 24
U, 22, 25
u, 22, 25
u8, 22, 25
prefix ++ and --
overloading, 356
prefix ++, 128
prefix --, 128
prefix unary operator function, see operator function, prefix unary
preprocessing, 463
preprocessing directive, 461–477
conditional inclusion, 463
error, 474
header inclusion, 465
import, 467
line control, 473
macro replacement, see macro, replacement
module, 466
null, 474
pragma, 474
source-file inclusion, 465
preprocessing-file, 461, 1649
preprocessing-op-or-punc, 19, 1631
preprocessing-operator, 19, 1631
preprocessing-token, 15, 1630
primary class template, see template, primary
primary equivalence class, 6
primary module interface unit, 248
primary-expression, 100, 1635
private, 19, 255, 289, see access control,
private, 461, 1646, 1647, 1649
private-module-fragment, 255, 1646
probability density function
cauchy_distribution, 1203
chi_squared_distribution, 1203
exponential_distribution, 1198
extreme_value_distribution, 1200
fisher_f_distribution, 1204
gamma_distribution, 1199
lognormal_distribution, 1202
normal_distribution, 1201
piecewise_constant_distribution, 1207
piecewise_linear_distribution, 1208
student_t_distribution, 1205
uniform_real_distribution, 1194
weibull_distribution, 1200
program, 54
ill-formed, 5
startup, 86–89
termination, 89–90
well-formed, 9, 11
program execution, 10–81
abstract machine, 10
as-if rule, see as-if rule
program semantics
affected by the existence of a variable or function definition, 422
projection, 7
promise object, 218
promise type, see coroutine, promise type
promoted integral type, 357
promotion
bool to int, 98
default argument promotion, 119
floating-point, 98
integral, 97
prospective destructor, see destructor, prospective
protected, 19, 289, see access control,
protected, 1647
protection, see access control, 504
prototype parameter
class, 402
provides storage, 59
prvalue, 92
pseudo-destructor, 103
ptr-abstract-declarator, 185, 1642
ptr-declarator, 184, 1642
ptr-operator, 184, 1642
ptrdiff_t, 140
implementation-defined type of, 140
public, 19, 289, see access control, public, 1647
punctuator, 19
pure-specifier, 263, 1646
purview
global module, 249
module unit, 249
named module, 249
Q
q-char, 17, 1631
q-char-sequence, 17, 1631
qualification
explicit, 47
qualified-id, 103, 1635
qualified-name-specifier, 228, 1644
<queue>, 907, 908
R
r-char, 24, 1634
r-char-sequence, 24, 1634
<random>, 1174, 1667
random number distribution
bernoulli_distribution, 1195
binomial_distribution, 1195
cauchy_distribution, 1203
chi_squared_distribution, 1203
discrete_distribution, 1205
exponential_distribution, 1198
extreme_value_distribution, 1200
gamma_distribution, 1199
gamma_distribution, 1199
geometric_distribution, 1196
lognormal_distribution, 1202
negative_binomial_distribution, 1197
negative_binomial_distribution, 1197
normal_distribution, 1201
piecewise_constant_distribution, 1207
piecewise_linear_distribution, 1208
poisson_distribution, 1197
requirements, 1180–1182
student_t_distribution, 1205
uniform_int_distribution, 1193
uniform_real_distribution, 1194
weibull_distribution, 1200
random number distributions
Bernoulli, 1195–1197
normal, 1201–1205
Poisson, 1197–1201
sampling, 1205–1210
uniform, 1193–1195
random number engine
linear_congruential_engine, 1183
mersenne_twister_engine, 1184
requirements, 1178–1179
subtract_with_carry_engine, 1185
with predefined parameters, 1189–1190
random number engine adaptor
discard_block_engine, 1186
independent_bits_engine, 1187
shuffle_order_engine, 1188
with predefined parameters, 1189–1190
random number generation, 1173–1210
distributions, 1193–1210
engines, 1182–1189
predefined engines and adaptors, 1189–1190
requirements, 1176–1182
synopsis, 1174–1176
utilities, 1191–1193
random number number generator, see uniform random
bit generator
random_device
implementation leeway, 1190
range, 930
counted, 930, 1035
<ranges>, 512, 598, 981, 986, 1658, 1693
<ratio>, 733, 1667
raw string literal, 24
raw_string, 24, 1634
reachable
declaration, 257
necessarily
translation unit, 257
translation unit, 257
reachable from, 930
ready, 1523, 1618
redefinition
typedef, 169
ref-qualifier, 184, 1642
reference, 76
assignment to, 147
call by, 119
forwarding, 439
lvalue, 76
null, 189
rvalue, 76
sizeof, 130
reference collapsing, 189
reference lifetime, 61
reference-compatible, 207
reference-related, 207
references
normative, 2
<regex>, 511, 978, 1508, 1667
regex_iterator
end-of-sequence, 1532
regex_token_iterator
end-of-sequence, 1534
regex_traits
specializations, 1515
region
declarative, 29, 35
intervening, 36
register, 19
register storage class, 1661
regular expression, 1506–1538
grammar, 1536
matched, 5
requirements, 1506
regular expression traits, 1536
char_class_type, 1507
isctype, 1507
lookup_classname, 1507, 1538
lookup_collatename, 1507
requirements, 1506, 1515
transform, 1507, 1538
transform_primary, 1507, 1538
translate, 1507, 1538
translate_nocase, 1507, 1538
reified object, see object, reified
reinterpret_cast, 19, see cast, reinterpret, 117, 413, 414, 1636
relational operator function, see operator function, relational
relational_expression, 141, 1638
relative path, see path, relative
relative_path, 1466
relaxed pointer safety, 68
release sequence, 82
remainder operator, see operator, remainder
remote time zone database, 1315
replacement
macro, see macro, replacement
replacement field
format string, 743
replacement_list, 462, 1650
representation
object, 73

Index
value, 73
represents the address, 77
requirement, 114
  compound, 115
  nested, 116
  simple, 115
  type, 115
requirement, 114, 1636
requirement-body, 114, 1636
requirement-parameter-list, 114, 1636
requirement-seq, 114, 1636
requirements, 479
container, 806, 829, 843, 844, 1523
  not required for unordered associated containers, 828
iterator, 929
numeric type, 1161
random number distribution, 1180–1182
random number engine, 1178–1179
regular expression traits, 1506, 1515
seed sequence, 1176–1177
sequence, 1523
uniform random bit generator, 1177
unordered associative container, 829
requires, 19, 113, 115, 116, 361, 1636, 1648
requires-clause, 361, 1648
trailing, 184
requires-expression, 113, 1636
rescanning and replacement, see macro, rescanning and replacement
reserved identifier, 18
reset, 649
reset
  random number distribution requirement, 1181
resolution, see overloading, resolution
restriction, 501, 502, 504, 1684
  address of bit-field, 284
  anonymous union, 287
bit-field, 284
destructor, 270
destructor, 277
extern, 167
local class, 288
operator overloading, 354
overloading, 354
pointer to bit-field, 284
reference, 189
static, 167
static member local class, 288
union, 286
result
gvalue, 93
prvalue, 93
result object, 93
result_type
  entity characterization based on, 1173
  random number distribution requirement, 1181
seed sequence requirement, 1177
rethrow, see exception handling, rethrow
return, 19, 160, 161, 1639
  and handler, 452
  and try block, 452
  constructor and, 161
  reference and, 207
return statement, see return
return type, 193
covariant, 293
overloading and, 325
return-type-requirement, 115, 1636
reversible container, see container, reversible
rewritten candidate, 332
right shift operator, see operator, right shift
root-directory, 1466
root-name, 1466
rounding, 98
rvalue, 92
lvalue conversion to, see conversion, lvalue-to-rvalue, 1673
rvalue reference, 188
S
s-char, 24, 1634
s-char-sequence, 24, 1633
safely-derived pointer, see pointer, safely-derived
  integer representation, 68
sample, 1113
sampling distributions, 1205–1210
satisfy, see constraint, satisfaction
scalar type, see type, scalar
scope, 1, 29, 35–41, 165
  anonymous union at namespace, 287
  block, 37
  class, 39
declarations and, 35–37
destructor and exit from, 160
enumeration, 40
exception declaration, 37
function, 38
function parameter, 38
function prototype, see scope, function parameter
global, 39
iteration-statement, 157
macro definition, see macro, scope of definition
name lookup and, 284
namespace, 38
overloading and, 327
potential, 35
selection-statement, 155
template parameter, 40
scope name hiding and, 41
scope resolution operator, see operator, scope resolution
scoped enumeration, see enumeration, scoped
Index

<scoped_allocator>, 510, 681, 1667
secondary comparison operator, 322
seed
random number engine requirement, 1178
seed sequence, 1176
requirements, 1176–1177
selected destructor, see destructor, selected
selection-statement, 155, 1639
semantics
class member, 119
<semaphore>, 512, 1611, 1658
sentinel, 930
separate compilation, see compilation, separate
separate translation, see compilation, separate
sequence constructor
seed sequence requirement, 1177
sequenced after, 80
sequenced before, 80
sequencing operator, see operator, comma
<set>, 510, 511, 868, 869, 978
<setjmp.h>, 551, 1683
shared lock, 1591
shared mutex types, 1591
shared state, see future, shared state
shared timed mutex type, 1592
<shared_mutex>, 512, 1586, 1665
shift operator
left, see operator, left shift
right, see operator, right shift
shift-expression, 140, 1637
short, 19, 176, 1641
typedef and, 166
shuffle_order_engine
generation algorithm, 1188
state, 1188
textual representation, 1189
transition algorithm, 1188
side effects, 11, 70, 80–82, 84, 155, 312, 318, 469,
504
visible, 84
sign, 23, 1633
signal, 81
signal-safe
__Exit, 521
abort, 521
evaluation, see evaluation, signal-safe
forward, 583
initializer_list functions, 537
memcpy, 802
memmove, 802
move, 583
move_if_noexcept, 583
numeric_limits members, 514
quick_exit, 522
signal, 552
type traits, 708
<signal.h>, 551, 1683
signature, 7, 8
signed, 19, 176, 1641
typedef and, 166
signed integer representation
ones’ complement, 128
two’s complement, 75, 222, 734, 1550, 1558
signed integer type, 74
signed-integer-class type, see type,
signed-integer-class
signed-integer-like, 937
significand, 24
similar types, 97
simple assignment operator function, see operator
function, simple assignment
simple call wrapper, 688
simple-capture, 108, 1635
simple-declaration, 164, 1640
simple-escape-sequence, 22, 1633
simple-requirement, 115, 1636
simple-template-id, 366, 1648
simple-type-specifier, 176, 1641
simply happens before, 83
size
seed sequence requirement, 1177
size_t, 130
sizeof, 19, 127, 413, 414, 1637
smart pointers, 658–672
source file, 13, 487, 500
source file character, see character, source file
<source_location>, 512, 531, 1658
<span>, 512, 915, 978, 1658
special member function, see constructor, see
assignment operator, see destructor
eligible, 269
specialization, 419
class template partial, 391
declared, 419
program-defined, 6
template, 418
template explicit, 427
specification
linkage, 237–240
extern, 237
implementation-defined, 237
nesting, 238
template argument, 432
specifications
C standard library exception, 504
C++, 505
specifier, 166–183
costeval, 170
costexpr, 170
constructor, 172
function, 171
constinit, 173
cv-qualifier, 174
declaration, 166
explicit, 168
friend, 170, 504
function, 168
inline, 173
static, 166
storage class, 166
type, see type specifier
typedef, 168
virtual, 168
specifier access, see access specifier
spherical harmonics \(Y^\ell_m\), 1243
\(<\text{stream}>\), 1427
stable algorithm, 8, 503
\(<\text{stack}>\), 907, 908
stack unwinding, 454
standard
structure of, 11
standard deviation
\(\text{normal\_distribution}\), 1201
standard integer type, 75
standard signed integer type, 74
standard unsigned integer type, 74
standard-layout class, see class, standard-layout
standard-layout struct, see struct, standard-layout
standard-layout type, see type, standard-layout
standard-layout union, see union, standard-layout
start
program, 88
startup
program, 488, 501
state, 624
discard\_block\_engine, 1186
\(\text{independent\_bits\_engine}\), 1187
linear\_congruential\_engine, 1183
\(\text{mersenne\_twister\_engine}\), 1184
\(\text{shuffle\_order\_engine}\), 1188
\(\text{subtract\_with\_carry\_engine}\), 1185
state entity, 687
statement, 154–163
continue in \(\text{for}\), 159
break, 160
compound, 155
continue, 160
declaration, 162
declaration in \(\text{for}\), 159
declaration in if, 154
declaration in \(\text{switch}\), 154, 157
declaration in \(\text{while}\), 158
do, 157, 158
empty, 155
expression, 155
fallthrough, 243
for, 157, 159
goto, 155, 160, 161
if, 155, 156
iteration, 157–160
jump, 160
labeled, 155
null, 155
range based \(\text{for}\), 159
selection, 155–157
\(\text{switch}\), 155, 157, 160
while, 157, 158
statement, 154, 1639
statement-seq, 155, 1639
static, 19, 166, 1640
destruction of local, 162
linkage of, 54, 167
overloading and, 325
static data member, see data member, static
static initialization, see initialization, static
static member, see member, static
static member function, see member function, static
static storage duration, 65
static type, see type, static
\(\text{static\_assert}\), 165
\(\text{static\_assert}\), 19, 164, 1640
not macro, 569
\(\text{static\_assert\_declaration}\), 164, 1640
\(\text{static\_cast}\), 117, 413, 414, 1636
\(\text{statically\_widen}\), 1245
\(<\text{stdalign\_h}>\), 1679, 1683, 1684
\(<\text{stdarg\_h}>\), 550, 1683
\(<\text{stdatomic\_h}>\)
absence thereof, 485, 1678
\(<\text{stdbool\_h}>\), 1673, 1679, 1683, 1684
\(<\text{stdint\_h}>\), 22, 25, 507, 508, 1679, 1683
\(<\text{stdexcept\_h}>\), 566
\(<\text{stdint\_h}>\), 521, 1505, 1683
\(<\text{stdio\_h}>\), 1504, 1683
\(<\text{stdlib\_h}>\), 508, 1683, 1684
\(<\text{stdnoreturn\_h}>\)
absence thereof, 485, 1678
stop request, 1573
stop state, 1573
\(<\text{stop\_token}>\), 511, 1574, 1658
storage class, 29
storage duration, 65–68
automatic, 65
class member, 68
dynamic, 65–68, 132
local object, 65
static, 65
thread, 65
storage management, see \(\text{delete}\), see \(\text{new}\)
storage-class-specifier, 166, 1640
stream
arbitrary-positional, 3
repositionable, 7
\(<\text{streambuf}>\), 1395
strict pointer safety, 68
string
distinct, 26
null terminator, 768
null-terminated byte, see NTBS
null-terminated character type, 6
null-terminated multibyte, see NTMBS
sizeof, 26
type of, 25
width, 746
Index

Suitable created object, see object, suitable created

SWAPPABLE WITH, 489
SWITCH, 19, 155, 157, 1639
and handler, 452
and try block, 452
SYMBOLIC LINK, 1458
Synchronize with, 82
SYNCTEMP, 512, 1453, 1658
SYNONYM, 228
type name as, 168
SYNTAX
class member, 119
synthesized three-way comparison, see three-way comparison, synthesized

SYNCSAVE, 571, 573, 1667

T
TARGET OBJECT, 687
TEMPLATE, 361–451
alias, 400
class, 381
deducible, 176
deducible arguments of, 337
function, 431
abbreviated, 194
equivalent, see equivalent, function templates
functionally equivalent, see functionally equivalent, function templates
key parameter of, 562
partial ordering, 397
member function, 382
primary, 391
static data member, 361
variable, 361

TEMPLATE INSTANTIATION, 418
template instantiation, 418

TEMPLATE NAME
linkage of, 362
template parameter, 30
template parameter object, 364
template parameter pack, 386
template parameter scope, 40
template-argument, 366, 1648
default, 365
template-argument-equivalent, 380
template-argument-list, 366, 1648
template-declaration, 361, 1647
template-head, 361, 1647
template-id, 366, 1648
valid, 368
template-name, 366, 1648
template-parameter, 362, 1648
template-parameter-list, 361, 1648
templated, 362
temporary, 69

sWAPPER, 490

suitable created object, see object, suitable created

SUMMARY
compatibility with ISO C, 1670
compatibility with ISO C++ 2003, 1665
compatibility with ISO C++ 2011, 1663
compatibility with ISO C++ 2014, 1660
compatibility with ISO C++ 2017, 1653
syntax, 1630

suitable created object, see object, suitable created

SUMMARY
compatibility with ISO C, 1670
compatibility with ISO C++ 2003, 1665
compatibility with ISO C++ 2011, 1663
compatibility with ISO C++ 2014, 1660
compatibility with ISO C++ 2017, 1653
syntax, 1630

suitable created object, see object, suitable created

SUMMARY
compatibility with ISO C, 1670
compatibility with ISO C++ 2003, 1665
compatibility with ISO C++ 2011, 1663
compatibility with ISO C++ 2014, 1660
compatibility with ISO C++ 2017, 1653
syntax, 1630

suitable created object, see object, suitable created

SUMMARY
compatibility with ISO C, 1670
compatibility with ISO C++ 2003, 1665
compatibility with ISO C++ 2011, 1663
compatibility with ISO C++ 2014, 1660
compatibility with ISO C++ 2017, 1653
syntax, 1630

Index 1759
constructor for, 70
destruction of, 70
destructor for, 70
elimination of, 69, 318
implementation-defined generation of, 69
order of destruction of, 70
terminate, 459, 460
called, 146, 454, 457, 459
termination
program, 87, 90
terminology
pointer, 77
text-line, 461, 1650
textual representation
discard_block_engine, 1187
independent_bits_engine, 1188
shuffle_order_engine, 1189
subtract_with_carry_engine, 1186
<tgmath.h>, 1678, 1683, 1684
this, 19, 100, 108, 268, 1635
type of, 268
this pointer, see this
thread, 81
<thread>, 511, 1578, 1667
thread of execution, 81
thread storage duration, see storage duration, thread
thread_local, 19, 166, 424, 427, 1640
threads
multiple, 81–86
<threads.h>
absence thereof, 485, 1678
three-way comparison
synthesized, 322
three-way comparison operator function, see
operator function, three-way comparison
throw, 3, 19, 146, 413, 1638
throw-expression, 146, 1638
throwing, see exception handling, throwing
<time.h>, 1334, 1683
timed mutex types, 1589
to-unsigned-like, 985
token, 16
alternative, 16
directive-introducing, 462
preprocessing, 15–16
token, 16, 1630
traceable pointer object, 68, 505
trailing requires-clause, see requires-clause, trailing
trailing-return-type, 184, 1642
traits, 8
transform
regular expression traits, 1507, 1538
transform_primary
regular expression traits, 1507, 1537, 1538
transition algorithm
discard_block_engine, 1187
independent_bits_engine, 1187
linear_congruential_engine, 1183
mersenne_twister_engine, 1184
shuffle_order_engine, 1188
subtract_with_carry_engine, 1185
translate
regular expression traits, 1507, 1538
translate_nocase
regular expression traits, 1507, 1538
translation
phases, 13–14
separate, see compilation, separate
translation unit, 13, 54
name and, 29
translation-unit, 54, 1634
transparently replaceable, 63
trigraph sequence, 1660
trivial class, see class, trivial
trivial type, see type, trivial
trivially copyable class, see class, trivially copyable
trivially copyable type, see type, trivially copyable
true, 19
truncation, 98
try, 19, 452
try block, see exception handling, try block
try-block, 452, 1649
TU-local
entity, 57
value or object, 57
<tuple>, 510–512, 582, 590, 598, 1667, 1692, 1693
tuple
and pair, 586
type, 29, 72–78
allocated, 131
arithmetic, 76
promoted, 357
array, 76
bitmask, 483
Boolean, 75
callable, 687
char, 75
char16_t, 22, 25, 75, 79
char32_t, 22, 25, 75, 79
char8_t, 75
c character, 75
c character container, 4
class and, 259
compound, 76
cost, 174
cv-combined, 97
cv-unqualified, 77
destination, 200
double, 76
dynamic, 4
enumerated, 76, 482, 483
example of incomplete, 73
extended integer, 75
extended signed integer, 74
extended unsigned integer, 74
float, 76
floating-point, 76
function, 76, 191, 192
fundamental, 76
implementation-defined sizeof, 74
implicit-lifetime, 74
incomplete, 31, 33, 37, 73, 95, 117–119, 121, 
127, 130, 131, 136, 289
incompletely-defined object, 73
int, 74
integer-class, 937
integral, 76
promoted, 357
literal, 74
long, 74
long double, 76
long long, 74
narrow character, 75
nodoscarg, 245
numeric, 1161
ordinary character, 75
over-aligned, 69
pointer, 76
polymorphic, 291
program-defined, 6
referenceable, 7
scalar, 74
short, 74
signed char, 74, 75
signed integer, 74
signed-integer-class, 937
similar, see similar types
standard integer, 75
standard signed integer, 74
standard unsigned integer, 74
standard-layout, 74
static, 8
structural, 364
trivial, 74
trivially copyable, 72, 74
underlying, 75
cchar16_t, 75, 97
cchar32_t, 75, 97
cchar8_t, 75
enumeration, 97, 222
fixed, 222
wchar_t, 75, 97
unsigned, 74
unsigned char, 74, 75
unsigned int, 74
unsigned integer, 74
unsigned long, 74
unsigned long long, 74
unsigned short, 74
unsigned integer-class, 937
void, 76
volatile, 174
wchar_t, 22, 25, 75, 79
type checking
argument, 119
type concept, see concept, type

type conversion, explicit, see casting
type generator, see template
type name, 184
nested, 285
scope of, 285
type pun, 126
type specifier
auto, 179
bool, 176
char, 176
cchar16_t, 176
cchar32_t, 176
cchar8_t, 176
const, 174
decltype, 178
decltype(auto), 179
double, 176
elaborated, 52, 176
enum, 176
float, 176
int, 176
long, 176
short, 176
signed, 176
simple, 175
unsigned, 176
void, 176
volatile, 174, 175
wchar_t, 176
type-constraint, 363, 1648
type-id, 185, 1642
type-id-only context, 403
type-name, 176, 1641
type-parameter, 363, 1648
type-parameter-key, 363, 1648
type-requirement, 115, 1636
type-specifier, 174, 1641
type-specifier-seq, 174, 1641
type_info, 122
<type_traits>, 510–512, 709, 1667, 1692
typedef
function, 193
typedef, 19, 166, 1640
overloading and, 326
typedef-name, 168, 1641
typedef, 19, 117, 122, 150, 413, 414, 1636
construction and, 317
destruction and, 317
<typeindex>, 735, 1667
<typeinfo>, 123, 530
typename, 19, 115, 176, 231, 337, 363, 402, 1636, 
1644, 1648
typename-specifier, 402, 1648
types
implementation-defined, 482
U
uchar.h>, 804, 1683
ud-suffix, 27, 1634
unary fold, 113
unary left fold, 113
unary operator
interpretation of, 354
overloaded, 354
unary right fold, 113
unary-expression, 127, 1637
unary-operator, 127, 1637
unblock, 8
undefined, 7, 498, 500, 501, 1221, 1225, 1228, 1388
undefined behavior, see behavior, undefined
unevaluated operand, 94
Unicode required set, 476
uniform distributions, 1193–1195
uniform random bit generator
requirements, 1177
uniform_int_distribution
discrete probability function, 1193
uniform_real_distribution
probability density function, 1194
union, 285
standard-layout, 261
union, 19, 76, 259, 285, 287, 1646
anonymous, 287
global anonymous, 287
union-like class, see class, union-like
unique pointer, 649
unit
translation, 487, 488, 500
universal character name, 13
universal-character-name, 14, 1630
Unix time, 1271
unnamed bit-field, 284
unnamed-namespace-definition, 225, 1644
unordered associative containers, 829
complexity, 828
equality function, 828
equivalent keys, 828, 829, 894, 903
exception safety, 840
hash function, 828
iterator invalidation, 839
iterators, 839
lack of comparison functions, 828
requirements, 828, 829, 839, 840
unique keys, 828, 829, 888, 899
unordered_set
unique keys, 899
unqualified-id, 102, 1635
unscoped enumeration, see enumeration, unscoped
unsequenced, 80
unsigned, 19, 176, 1641
typedef and, 166
unsigned integer type, 74
unsigned-integer-class type, see type,
unsigned-integer-class
unsigned-integer-like, 937
unsigned-suffix, 20, 1632
unspecified, 524, 525, 530, 1117, 1432, 1686, 1688
unspecified behavior, see behavior, unspecified
unwinding
stack, 454
uppercase, 18, 484
upstream, 679
upstream allocator, 677
usable
binary operator expression, 320
usable candidate, see candidate, usable
usable in constant expressions, 148
user-defined conversion sequence, see conversion
sequence, user-defined
user-defined literal, see literal, user-defined
overloaded, 359
user-defined-character-literal, 27, 1634
user-defined-floating-point-literal, 26, 1634
user-defined-integer-literal, 26, 1634
user-defined-literal, 26, 1634
user-defined-string-literal, 27, 1634
user-provided, 216
uses-allocator construction, 644
using, 19, 164, 226, 229, 231, 240, 1640, 1644, 1645
using-declaration, 231–237
using-declaration, 231, 1644
using-declarator, 231, 1644
using-declarator-list, 231, 1644
using-directive, 229–231
using-directive, 229, 1644
using-enum-declaration, 224, 1644
usual arithmetic conversions, see conversion, usual
arithmetic
usual deallocation function, 67
UTF-16 character literal, 22
UTF-16 string literal, 25
UTF-32 character literal, 22
UTF-32 string literal, 25
UTF-8 character literal, 22
UTF-8 string literal, 25
<utility>, 490, 510–512, 580, 598, 1667, 1684, 1693
va-opt-replacement, 469, 1650
vacuous initialization, see initialization, vacuous
<valarray>, 1210, 1212
valid but unspecified state, 9
value, 73
call by, 119
denormalized, see number, subnormal
indeterminate, 64
invalid pointer, 77
null member pointer, 99
null pointer, 77, 98
undefined unrepresentable integral, 98
value category, 92
value computation, 70, 80–81, 84, 120, 134, 146
value type, 929
value-initialization, 199
variable, 29
function-local predefined, 214
indeterminate uninitialized, 199
inline, 173
local, 37
needed for constant evaluation, 153
program semantics affected by the existence of a variable definition, 422
variable arguments, 469
variable template
definition of, 361
<variant>, 512, 612, 1662, 1693
variant member, 287
<vector>, 510, 511, 840, 842, 979
vectorization-unsafe, 1046
.VERSION>, 510, 1658
_virt-specifier_, 263, 1646
_virt-specifier-seq_, 263, 1646
virtual, 19, 168, 277, 289, 385, 1641, 1647
virtual base class, see base class, virtual, see base class, virtual
virtual function, see function, virtual, see function, virtual
virtual function call, 295
constructor and, 317
destructor and, 317
undefined pure, 296
visibility, 41
visible, 41
visible side effects, see side effects, visible
void, 19, 176, 1641
void*
type, 77
void&*, 188
volatile, 19, 77, 184, 1642
constructor and, 269, 270
destructor and, 269, 277
implementation-defined, 175
overloading and, 326
volatile member function, 268
volatile object, see object, volatile
volatile-qualified, 77

W
waiting function, see function, waiting
_wchar.h>, 803, 1683
_wchar_t, 19, see type, wchar_t, 176, 1641
_wctype.h>, 801, 1683
weakly parallel forward progress guarantees, 86
weibull_distribution
probability density function, 1200
weights
discrete_distribution, 1205
piecewise_constant_distribution, 1207
weights at boundaries
piecewise_linear_distribution, 1208
well-formed program, see program, well-formed
while, 19, 157, 159, 1639
whitespace, 15, 16
wide string literal, 25
wide-character, 22
null, 15
wide-character literal, 22
wide-character set
basic execution, 15
exeguion, 15
width, 74, 283, 746
worse conversion sequence, see conversion sequence, worse

X
xor, 19
xor_eq, 19
xvalue, 92

Y
Y^m_\ell_ (spherical associated Legendre functions), 1243
yield-expression, 145, 1638

Z
zero
division by undefined, 91
remainder undefined, 91
undefined division by, 139
zero-initialization, 199
zeta functions \(\zeta\), 1242

Index
The index of grammar productions lists page numbers where each grammar production is defined. The first bold page number is the page in the general text where the grammar production is defined. The second bold page number is the corresponding page in the Grammar summary (Annex A). Other page numbers refer to pages where the grammar production is mentioned in the general text.

Index of grammar productions

abstract-declarator, 185, 185, 192, 195, 1642
abstract-pack-declarator, 185, 1643
access-specifier, 289, 289, 301, 302, 1647
additive-expression, 139, 1637
alias-declaration, 30, 56, 164, 169, 192, 250, 254, 264, 361, 381, 400, 1640
alignment-specifier, 240, 241, 242, 387, 1645
and-expression, 143, 1638
asm-declaration, 150, 164, 237, 237, 1645
assignment-expression, 147, 165, 179, 181, 195, 203, 206, 220, 283, 309, 335, 1638
assignment-operator, 147, 1638
attribute, 240, 241, 387, 1645
attribute-argument-clause, 240, 240–246, 1645
attribute-declaration, 30, 164, 165, 1640
attribute-list, 240, 240–246, 387, 1645
attribute-namespaces, 240, 240, 241, 1645
attribute-scoped-token, 240, 240, 1645
attribute-specifier, 240, 240, 241, 1645
attribute-token, 19, 240, 240–246, 464, 500, 1645
attribute-using-prefix, 240, 240, 1645
await-expression, 128, 128, 129, 150, 217, 218, 550, 1637
balanced-token, 240, 1645
balanced-token-seq, 240, 241, 1645
base-clause, 259, 288, 1647
base-specifier, 43, 49, 289, 289, 297, 300, 304, 387, 1647
base-specifier-list, 276, 288, 289, 300, 312, 314, 321, 387, 388, 1647
binary-digit, 20, 20, 1632
binary-exponent-part, 23, 24, 1633
binary-literal, 20, 21, 1632
block-declaration, 164, 1640
boolean-literal, 26, 1634
brace-or-equal-initializer, 79, 198, 198, 200, 203, 264, 265, 271, 283, 413, 1643
braced-init-list, 70, 79, 117, 119, 132, 147, 153, 161, 181, 182, 198, 200, 204, 209, 210, 212, 309, 310, 313, 337, 408, 413, 1643, 1652, 1661
c-char, 21, 22, 25, 1633
c-char-sequence, 14, 21, 1633
capture, 108, 387, 1645
capture-default, 32, 108, 109, 111, 1635, 1655, 1681
capture-list, 108, 387, 1645
character-literal, 13, 14, 21, 22, 23, 26, 464, 465, 469, 471, 1345, 1632, 1664, 1671
class-head, 36, 241, 259, 259, 1646
class-head-name, 227, 259, 259, 367, 1646
class-key, 52, 165, 178, 227, 259, 259, 261, 262, 285, 302, 382, 1646
class-name, 12, 37, 41, 52, 54, 165, 169, 170, 177, 178, 259, 259, 263, 276, 1646
class-or-decltype, 259, 289, 289, 310, 367, 403, 1647
class-specifier, 36, 57, 165, 174, 259, 259, 264, 265, 269, 1646
class-virt-specifier, 259, 259, 718, 1646
cast-expression, 140, 1638
compound-declaration, 115, 115, 116, 1636
concept-definition, 30, 401, 401, 402, 1648
concept-name, 57, 254, 369, 401, 401, 1648
condition, 38, 154, 154, 155, 157, 159, 162, 200, 453, 1639
conditional-expression, 4, 144, 144, 1638
conditionally-supported-directive, 461, 462, 469, 1650
constraint-expression, 102, 116, 184, 327, 362, 363, 369, 375, 377–379, 397, 402, 481, 1648
constraint-logical-and-expression, 361, 1648
constraint-logical-or-expression, 184, 361, 362, 1648
coroutine-return-statement, 161, 217, 1639
coroutine-initializer, 44, 213, 214, 271, 309, 309–311, 313, 319, 453, 454, 1647
276, 282, 296, 297, 303, 307, 367, 390, 403, 411, 413, 436, 444, 1635
nested-namespace-definition, 225, 226, 1644
nested-requirement, 115, 116, 116, 1636
new-declarator, 131, 131, 1637
new-initializer, 72, 131, 131, 132, 134, 135, 200, 271, 1637
new-line, 462, 467, 1650
new-placement, 131, 134, 135, 1637
new-type-id, 78, 131, 131, 132, 180, 183, 403, 413, 1637
nodeclspec-function-declaration, 164, 164, 165, 1640
noexcept-expression, 131, 131, 1637
nondigit, 18, 466, 1631
nonzero-digit, 20, 1632
noptptr-abstract-declarator, 185, 1642
noptptr-abstract-pack-declarator, 185, 1643
noptptr-declarator, 184, 1642
noptptr-new-declarator, 131, 132, 1637
octal-digit, 20, 20, 1632
octal-escape-sequence, 22, 1633
octal-literal, 20, 21, 1632
opaque-enum-declaration, 30, 36, 221, 221, 222, 227, 264, 1644
operator, 353, 1647
operator-function-id, 29, 102, 353, 353, 354, 362, 380, 424, 1647
operator-or-punctuator, 19, 19, 1631
parameter-declaration, 30, 168, 179, 180, 183, 192, 192, 194, 195, 217, 242, 363, 366, 386, 403, 446, 1643
parameter-declaration-list, 192, 438, 444, 1643
parameters-and-qualifiers, 184, 1642
pathname, 1466
placeholder-type-specifier, 176, 177, 179, 179, 182, 194, 1642
pm-expression, 138, 138, 1637
pointer-literal, 26, 1634
postfix-expression, 42, 45, 49, 53, 54, 103, 117, 118, 120, 254, 267, 331, 355, 408, 415, 458, 532, 685, 951, 1044, 1636
pp-global-module-fragment, 461, 462, 1649
pp-import, 462, 467, 467, 1650
pp-module, 466, 466, 1650
pp-number, 17, 463, 464, 1631, 1660, 1663
pp-private-module-fragment, 461, 1649
pp-tokens, 462, 463, 464, 467, 470, 476, 1650
preferred-separator, 1465, 1466, 1466, 1467, 1470, 1471, 1473
preprocessing-file, 461, 1649
preprocessing-op-or-punc, 19, 1631
preprocessing-operator, 19, 1631
preprocessing-token, 15, 16, 463, 1630, 1650
primary-expression, 100, 101, 331, 1635
private-module-fragment, 33, 57, 180, 249, 252, 255, 255–257, 415, 1646
ptr-abstract-declarator, 185, 1642
ptr-declarator, 184, 270, 276, 1642
ptr-operator, 184, 1642
pure-specifier, 263, 264, 265, 277, 295, 296, 1646
q-char, 17, 1631
q-char-sequence, 17, 17, 1631
qualified-name-specifier, 228, 228, 1644
r-char, 15, 24, 1634
r-char-sequence, 14, 24, 1634
raw-string, 24, 24, 25, 1634
ref-qualifier, 7, 8, 122, 139, 184, 188, 189, 191–193, 215, 220, 235, 268, 269, 320, 326, 329, 350, 357, 398, 1642
relational-expression, 141, 1638
relative-path, 1466, 1466, 1476
replacement-list, 462, 468, 1650
requirement, 114, 114, 115, 1636
requirement-body, 114, 114, 1636
requirement-parameter-list, 114, 114, 403, 1636
requirement-seq, 114, 1636
requires-clause, 7, 8, 102, 104–106, 114, 184, 193, 235, 282, 293, 321, 325–327, 361, 362, 374, 377, 381, 390, 397, 404, 420, 424, 967, 1002, 1010, 1648, 1653
requires-expression, 113, 113–115, 553, 1636, 1653
return-type-requirement, 115, 115, 1636
root-directory, 1463, 1466, 1466, 1467, 1474, 1477
root-name, 1463, 1466, 1466, 1467, 1470, 1474, 1476
s-char, 24, 1634
s-char-sequence, 14, 24, 1633
selection-statement, 154, 155, 155, 1639
shift-expression, 140, 1637
sign, 23, 24, 1633, 1660
simple-declaration, 128, 164, 164, 164, 183, 403, 424, 1640
simple-escape-sequence, 22, 1633
simple-requirement, 115, 115, 116, 1636
Index of library headers

The bold page number for each entry refers to the page where the synopsis of the header is shown.

<algorithm>, 510, 512, 1049, 1667
<any>, 510, 623, 1662
<array>, 510–512, 598, 840, 840, 843, 978, 1667, 1693
<assert.h>, 487, 569, 1679, 1683
.atomic, 487, 510, 1539, 1667, 1702
<assert.h>, 487, 569, 1679, 1683
.bitset, 628
<assert.h>, 487, 569, 1679
<cassert>, 801, 1344
<cerrno>, 500, 569, 573
<cfenv>, 1161, 1162, 1667
<cfloat>, 510, 520, 520
<charconv>, 512, 739, 1658, 1662
<chrono>, 510, 1245, 1667
<cinttypes>, 1504, 1505, 1667
<climits>, 58, 510, 519, 1687
<locale>, 484, 1374, 1679
<cmath>, 511, 1229, 1237, 1658, 1684
<codecvt>, 1667, 1696
<compare>, 141, 512, 538, 1658
<complex>, 510, 1162, 1163, 1658, 1683, 1684
<complex.h>, 1678, 1683, 1683, 1684
<concepts>, 510, 554, 1658
<condition_variable>, 1604, 1667
<coroutine>, 511, 546, 547, 1658
<csetjmp>, 500, 550, 551, 1680
<csignal>, 550, 551
<cstdlib>, 192, 500, 550, 550
<cstddef>, 130, 140, 507, 510, 1679, 1680
<cstdint>, 75, 520, 520, 1505, 1549, 1557, 1667
<cstdio>, 522, 1378, 1379, 1385, 1444, 1503, 1504, 1679
<cstring>, 89, 90, 487, 507, 508, 521, 522, 525, 550, 633, 649, 804, 1160, 1210, 1237, 1355, 1679, 1684
<csttl>, 268, 484, 802, 1679, 1687, 1691
<ctime>, 1334, 1334, 1336, 1679
<ctype.h>, 801, 1683
<cstdio>, 500, 804, 804, 1667, 1678
<cwchar>, 500, 763, 802, 803, 804, 1679
<cwctype>, 500, 801
<deque>, 510, 511, 840, 841, 978
<errno.h>, 569, 1683
<exception>, 512, 533
<execution>, 511, 737, 738, 1662
<fenv.h>, 1162, 1683
<filesystem>, 510, 511, 1459, 1662
<float.h>, 520, 1683
<format>, 511, 741, 1658
<forward_list>, 510, 511, 840, 841, 978, 1667
<fstream>, 1441, 1441, 1501
<functional>, 510–512, 685
<future>, 1616, 1667
<initializer_list>, 537, 1667
<inttypes.h>, 1505, 1683
<iomanip>, 512, 1403, 1423, 1424
<iostream>, 1379
<iosfwd>, 5, 1376, 1377
<iosstream>, 405, 1378, 1378, 1385
<locale.h>, 486, 1679, 1683, 1683, 1684
<locale>, 510, 1403, 1404
<iterator>, 510–512, 922, 978, 986, 1693
<latch>, 511, 1613, 1658
<limits>, 510, 512
<limits.h>, 519, 1683
<list>, 510, 511, 840, 842, 978
<locale>, 510, 1335, 1336, 1698
<locale.h>, 1374, 1683
<map>, 510, 511, 868, 868, 978
<math.h>, 1237, 1683
<memory>, 510–512, 633, 634, 1154, 1694
<memory_resource>, 511, 672, 1662
<mutex>, 512, 1586, 1667
<new>, 66, 511, 523
<numbers>, 511, 1244, 1658
<numerics>, 511, 512, 1141
<optional>, 511, 600, 1662
<ostream>, 510, 1403, 1415
<queue>, 907, 908
<random>, 1174, 1667
<ranges>, 512, 598, 981, 986, 1658, 1693
<ratio>, 733, 1667
<regex>, 511, 978, 1508, 1508, 1667
<scoped_allocator>, 510, 681, 1667
<semaphore>, 512, 1611, 1658
<set>, 510, 511, 868, 869, 978
<setjmp.h>, 551, 1683
<shared_mutex>, 512, 1586, 1665
<signal.h>, 551, 1683
<source_location>, 512, 531, 1658
<span>, 512, 915, 915, 978, 1658
<sstream>, 1427, 1427
<stack>, 907, 908
<stdalign.h>, 1679, 1683, 1683, 1684
<stdarg.h>, 550, 1683
<stdbool.h>, 1673, 1679, 1683, 1683, 1684
<stddef.h>, 22, 25, 507, 508, 1679, 1683
<stdexcept>, 566
<stdint.h>, 521, 1505, 1683
<stdio.h>, 1504, 1683
<stdlib.h>, 508, 1683, 1684
<stop_token>, 511, 1574, 1658
<streambuf>, 1395, 1395
<string>, 510–512, 762, 765, 978
<string.h>, 802, 1683
<string_view>, 510–512, 791, 979, 1662
<strstream>, 1685
<syncstream>, 512, 1453, 1453, 1658
<system_error>, 571, 573, 1667
<tgmath.h>, 1678, 1683, 1684, 1684
<thread>, 511, 1578, 1667
<time.h>, 1334, 1683
<tuple>, 510–512, 582, 590, 598, 1667, 1692, 1693
<typename_traits>, 510–512, 709, 1667, 1692
<typeindex>, 735, 1667
<typeinfo>, 123, 530
<uchar.h>, 804, 1683
<unordered_map>, 510–512, 886, 886, 979, 1667
<unordered_set>, 510, 511, 886, 887, 979, 1667
<utility>, 490, 510–512, 580, 598, 1667, 1684, 1693
<valarray>, 1210, 1212
<variant>, 512, 612, 1662, 1693
<vector>, 510, 511, 840, 842, 979
<version>, 510, 510, 1658
<wchar.h>, 803, 1683
<wctype.h>, 801, 1683
Symbols

_Exit, 507, 521
_IOFBF, 1503
_IOLBF, 1503
_INBF, 1503
__alignas_is_defined, 1683
__bool_true_false_are_defined, 1683
__cpp_lib_addressof constexpr, 510
__cpp_lib_allocator_traits_is_always_equal, 510
__cpp_lib_any, 510
__cpp_lib_apply, 510
__cpp_lib_array constexpr, 510
__cpp_lib_as constexpr, 510
__cpp_lib_assume_aligned, 510
__cpp_lib_atomic_flag_test, 510
__cpp_lib_atomic_float, 510
__cpp_lib_atomic_is_always_lock_free, 510
__cpp_lib_atomic_lock_free_type_aliases, 510
__cpp_lib_atomic_ref, 510
__cpp_lib_atomic_shared_ptr, 510
__cpp_lib_atomic_value_initialization, 510
__cpp_lib_atomic_wait, 510
__cpp_lib_array constexpr, 510
__cpp_lib_array constexpr, 510
__cpp_lib_array constexpr, 510
__cpp_lib_array constexpr, 510
__cpp_lib_array constexpr, 510
__cpp_lib_addressof constexpr, 510
__cpp_lib_allocator_traits_is_always_equal, 510
__cpp_lib_any, 510
__cpp_lib_apply, 510
__cpp_lib_array constexpr, 510
__cpp_lib_as constexpr, 510
__cpp_lib_assume_aligned, 510
__cpp_lib_atomic_flag_test, 510
__cpp_lib_atomic_float, 510
__cpp_lib_atomic_is_always_lock_free, 510
__cpp_lib_atomic_lock_free_type_aliases, 510
__cpp_lib_atomic_ref, 510
__cpp_lib_atomic_shared_ptr, 510
__cpp_lib_atomic_value_initialization, 510
__cpp_lib_atomic_wait, 510
__cpp_lib_array constexpr, 510
__cpp_lib_array constexpr, 510
__cpp_lib_array constexpr, 510
__cpp_lib_array constexpr, 510
__cpp_lib_array constexpr, 510
__cpp_lib_addressof constexpr, 510
<table>
<thead>
<tr>
<th>Library Name</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_lib_ranges</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_raw_memory_algorithms</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_remove_cvref</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_result_of_sFINAE</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_robust_nonmodifying_seq_ops</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_sample</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_scoped_lock</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_semaphore</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_shared_mutex</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_shared_ptr_arrays</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_shared_ptr_weak_type</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_shared_timed_mutex</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_shift</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_smart_ptr_for_overwrite</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_source_location</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_span</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_ssize</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_starts_ends_with</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_string_udls</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_string_view</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_syncbuf</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_three_way_comparison</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_to_address</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_to_array</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_to_chars</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_transformation_trait_aliases</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_transparent_operators</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_tuple_element_t</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_tuples_by_type</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_type_identity</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_type_trait_variable_templates</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_uncaught_exceptions</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_unordered_map_try_emplace</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_unwrap_ref</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_variant</td>
<td>512</td>
</tr>
<tr>
<td>__cpp_lib_void_t</td>
<td>512</td>
</tr>
</tbody>
</table>

**Numbers**

<table>
<thead>
<tr>
<th>Number</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>701</td>
</tr>
</tbody>
</table>

**A**

cauchoy_distribution | 1204

extreme_value_distribution | 1201

uniform_int_distribution | 1194

uniform_real_distribution | 1194

weibull_distribution | 1200

abbrev

sys_info | 1318

abort | 90, 160, 487, 507, 521, 528, 535

abs | 507, 1229, 1237, 1504

complex | 1168

duration | 1266

valarray | 1221

absolute | 1491

accumulate | 1145

acos | 1229

complex | 1168

valarray | 1221

acosf | 1229

acossh | 1229

acoshl | 1229

acosl | 1229

acq_rel

memory_order | 1543

acquire

counting_semaphore | 1612

memory_order | 1543

add_const | 726

add_const_t | 711

add_cv | 726

add_cv_t | 711

add_lvalue_reference | 726

add_lvalue_reference_t | 711

add_pointer | 727

add_pointer_t | 711

add_rvalue_reference | 726

add_rvalue_reference_t | 711

add_volatile | 726

add_volatile_t | 711

address

coroutine_handle | 548

coroutine_handle<noop_coroutine_promise> | 550

addressof | 649

adjacent_difference | 1152

adjacent_find | 1092

adopt_lock | 1594

adopt_lock_t | 1594

advance | 950, 951

subrange | 997

advance_to

basic_format_context | 755

basic_format_parse_context | 753

align | 643

align_val_t | 523

aligned_alloc | 507, 649, 1680

aligned_storage | 728, 729

aligned_storage_t | 712

aligned_union | 728

aligned_union_t | 712

alignment_of | 723

alignment_of_v | 714

all | 1009

bitset | 632

all_of | 1087

all_t | 981

allocate

class | 648

allocator_traits | 647

memory_resource | 673

class | 675

Index of library names | 1772
scoped_allocator_adaptor, 684
allocate_bytes
polymorphic_allocator, 675
allocate_object
polymorphic_allocator, 675
allocate_shared, 664–666
allocator, 648
allocate, 648
deallocate, 648
difference_type, 648
is_always_equal, 648
operator=, 648
operator==, 649
propagate_on_container_move_assignment, 648
size_type, 648
value_type, 648
allocator_arg, 643
allocator_arg_t, 643
allocator_traits, 646
allocate, 647
const_pointer, 647
const_void_pointer, 647
construct, 647
deallocate, 647
destroy, 648
difference_type, 647
is_always_equal, 647
max_size, 648
pointer, 646
propagate_on_container_copy_assignment, 647
propagate_on_container_move_assignment, 647
propagate_on_container_swap, 647
rebind_alloc, 647
select_on_container_copy_construction, 648
size_type, 647
void_pointer, 647
allocator_type
basic_string, 768
alpha
gamma_distribution, 1199
always_noconv
codecvt, 1349
ambiguous
local_info, 1318
ambiguous_local_time, 1316
constructor, 1317
any
constructor, 624, 625
destructor, 625
emplace, 626
has_value, 627
operator=, 625, 626
reset, 626
swap, 626, 627
type, 627
any (member)
bitset, 632
any_cast, 627, 628
any_of, 1087
append
basic_string, 778
path, 1471
apply, 597
valarray, 1219
arg, 1170
basic_format_context, 754
complex, 1168
argument_type
zombie, 499
array, 843, 845
begin, 843
data, 844
end, 843
fill, 844
get, 845
max_size, 843
size, 843, 844
swap, 844
arrive
barrier, 1615
arrive_and_drop
barrier, 1616
arrive_and_wait
barrier, 1616
latch, 1614
as_bytes, 921
as_const, 584
as_writable_bytes, 921
asctime, 1334
asin, 1229
complex, 1168
valarray, 1221
asinf, 1229
asinh, 1229
complex, 1169
asinhf, 1229
asinhl, 1229
assign
assignable_from, 558
assoc_laguerre, 1238
assoc_laguerref, 1238
assoc_laguerrel, 1238
assoc_legendre, 1239
Index of library names
Index of library names
Index of library names

\begin{itemize}
\item \texttt{atomic\_ref\langle integral\rangle, 1549}
\item \texttt{atomic\_ref\langle T\rangle, 1551}
\item \texttt{atomic\_store\_explicit, 1554}
\item \texttt{atomic\_thread\_fence, 1569}
\item \texttt{atomic\_uchar, 1543}
\item \texttt{atomic\_uint, 1543}
\item \texttt{atomic\_uint16\_t, 1543}
\item \texttt{atomic\_uint32\_t, 1543}
\item \texttt{atomic\_uint64\_t, 1543}
\item \texttt{atomic\_uint8\_t, 1543}
\item \texttt{atomic\_uint_fast16\_t, 1543}
\item \texttt{atomic\_uint_fast32\_t, 1543}
\item \texttt{atomic\_uint_fast64\_t, 1543}
\item \texttt{atomic\_uint_fast8\_t, 1543}
\item \texttt{atomic\_uint_least16\_t, 1543}
\item \texttt{atomic\_uint_least32\_t, 1543}
\item \texttt{atomic\_uint_least64\_t, 1543}
\item \texttt{atomic\_uint_least8\_t, 1543}
\item \texttt{atomic\_uintmax\_t, 1543}
\item \texttt{atomic\_uintptr\_t, 1543}
\item \texttt{atomic\_ulong, 1543}
\item \texttt{atomic\_ulong, 1543}
\item \texttt{atomic\_unsigned\_lock\_free, 1543}
\item \texttt{atomic\_ushort, 1543}
\item \texttt{ATOMIC\_VAR\_INIT, 1703}
\item \texttt{atomic\_wchar\_t, 1543}
\item \texttt{ATOMIC\_WCHAR\_T\_LOCK\_FREE, 1545}
\item \texttt{auto\_ptr}
\item \texttt{zombie, 498}
\item \texttt{auto\_ptr\_ref}
\item \texttt{zombie, 498}
\item \texttt{await\_ready}
\item \texttt{suspend\_always, 550}
\item \texttt{suspend\_never, 550}
\item \texttt{await\_resume}
\item \texttt{suspend\_always, 550}
\item \texttt{suspend\_never, 550}
\item \texttt{await\_suspend}
\item \texttt{suspend\_always, 550}
\item \texttt{suspend\_never, 550}
\item \texttt{awk}
\item \texttt{syntax\_option\_type, 1512}
\end{itemize}

\textbf{B}

\begin{itemize}
\item \texttt{cauchy\_distribution, 1204}
\item \texttt{extreme\_value\_distribution, 1201}
\item \texttt{uniform\_int\_distribution, 1194}
\item \texttt{uniform\_real\_distribution, 1195}
\item \texttt{weibull\_distribution, 1200}
\end{itemize}

\textbf{back}

\begin{itemize}
\item \texttt{basic\_string, 777}
\item \texttt{basic\_string\_view, 796}
\item \texttt{span, 920}
\item \texttt{view\_interface, 994}
\end{itemize}

\textbf{back\_insert\_iterator}

\begin{itemize}
\item \texttt{constructor, 957}
\item \texttt{operator*, 957}
\end{itemize}
<table>
<thead>
<tr>
<th>Operator/Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator++</code></td>
<td>958</td>
</tr>
<tr>
<td><code>operator=</code></td>
<td>957</td>
</tr>
<tr>
<td><code>back_inserter</code></td>
<td>958</td>
</tr>
<tr>
<td><code>bad</code></td>
<td>1392</td>
</tr>
<tr>
<td><code>bad_alloc</code></td>
<td>134, 524, 528</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>528</td>
</tr>
<tr>
<td><code>what</code></td>
<td>528</td>
</tr>
<tr>
<td><code>bad_any_cast</code></td>
<td>623</td>
</tr>
<tr>
<td><code>what</code></td>
<td>624</td>
</tr>
<tr>
<td><code>bad_array_new_length</code></td>
<td>528</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>528</td>
</tr>
<tr>
<td><code>what</code></td>
<td>528</td>
</tr>
<tr>
<td><code>bad_cast</code></td>
<td>121, 530, 531</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>531</td>
</tr>
<tr>
<td><code>what</code></td>
<td>531</td>
</tr>
<tr>
<td><code>bad_exception</code></td>
<td>534</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>534</td>
</tr>
<tr>
<td><code>what</code></td>
<td>534</td>
</tr>
<tr>
<td><code>bad_function_call</code></td>
<td>701</td>
</tr>
<tr>
<td><code>what</code></td>
<td>701</td>
</tr>
<tr>
<td><code>bad_optional_access</code></td>
<td>609</td>
</tr>
<tr>
<td><code>bad_typeof</code></td>
<td>122, 530, 531</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>531</td>
</tr>
<tr>
<td><code>what</code></td>
<td>531</td>
</tr>
<tr>
<td><code>bad_variant_access</code></td>
<td>622</td>
</tr>
<tr>
<td><code>what</code></td>
<td>623</td>
</tr>
<tr>
<td><code>bad_weak_ptr</code></td>
<td>658</td>
</tr>
<tr>
<td><code>what</code></td>
<td>658</td>
</tr>
<tr>
<td><code>barrier</code></td>
<td>1615</td>
</tr>
<tr>
<td><code>arrive</code></td>
<td>1615</td>
</tr>
<tr>
<td><code>arrive_and_drop</code></td>
<td>1616</td>
</tr>
<tr>
<td><code>arrive_and_wait</code></td>
<td>1616</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>1615</td>
</tr>
<tr>
<td><code>max</code></td>
<td>1615</td>
</tr>
<tr>
<td><code>wait</code></td>
<td>1615</td>
</tr>
<tr>
<td><code>base</code></td>
<td>1036</td>
</tr>
<tr>
<td><code>common_view</code></td>
<td>1036</td>
</tr>
<tr>
<td><code>counted_iterator</code></td>
<td>970</td>
</tr>
<tr>
<td><code>drop_view</code></td>
<td>1023</td>
</tr>
<tr>
<td><code>drop_while_view</code></td>
<td>1024</td>
</tr>
<tr>
<td><code>elements_view</code></td>
<td>1039</td>
</tr>
<tr>
<td><code>elements_view::iterator</code></td>
<td>1041</td>
</tr>
<tr>
<td><code>filter_view</code></td>
<td>1010</td>
</tr>
<tr>
<td><code>filter_view::iterator</code></td>
<td>1012</td>
</tr>
<tr>
<td><code>filter_view::sentinel</code></td>
<td>1013</td>
</tr>
<tr>
<td><code>join_view</code></td>
<td>1025</td>
</tr>
<tr>
<td><code>move_iterator</code></td>
<td>961</td>
</tr>
<tr>
<td><code>move_sentinel</code></td>
<td>964</td>
</tr>
<tr>
<td><code>reverse_iterator</code></td>
<td>954</td>
</tr>
<tr>
<td><code>reverse_view</code></td>
<td>1037</td>
</tr>
<tr>
<td><code>split_view</code></td>
<td>1030</td>
</tr>
<tr>
<td><code>take_view</code></td>
<td>1019</td>
</tr>
<tr>
<td><code>take_view::sentinel</code></td>
<td>1021</td>
</tr>
<tr>
<td><code>take_while_view</code></td>
<td>1021</td>
</tr>
<tr>
<td><code>transform_view</code></td>
<td>1013</td>
</tr>
<tr>
<td><code>transform_view::iterator</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>transform_view::sentinel</code></td>
<td>1019</td>
</tr>
<tr>
<td><code>basic</code></td>
<td>1376, 1441</td>
</tr>
<tr>
<td><code>syntax_option_type</code></td>
<td>1512</td>
</tr>
<tr>
<td><code>basic_common_reference</code></td>
<td>728</td>
</tr>
<tr>
<td><code>basic_filebuf</code></td>
<td>1441</td>
</tr>
<tr>
<td><code>close</code></td>
<td>1444</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>1443</td>
</tr>
<tr>
<td><code>destructor</code></td>
<td>1443</td>
</tr>
<tr>
<td><code>imbue</code></td>
<td>1447</td>
</tr>
<tr>
<td><code>is_open</code></td>
<td>1444</td>
</tr>
<tr>
<td><code>open</code></td>
<td>1444</td>
</tr>
<tr>
<td><code>operator=</code></td>
<td>1443</td>
</tr>
<tr>
<td><code>overflow</code></td>
<td>1446</td>
</tr>
<tr>
<td><code>pbackfail</code></td>
<td>1445</td>
</tr>
<tr>
<td><code>seekoff</code></td>
<td>1446</td>
</tr>
<tr>
<td><code>seekpos</code></td>
<td>1446</td>
</tr>
<tr>
<td><code>setbuf</code></td>
<td>1446</td>
</tr>
<tr>
<td><code>showmanyc</code></td>
<td>1445</td>
</tr>
<tr>
<td><code>swap</code></td>
<td>1443</td>
</tr>
<tr>
<td><code>sync</code></td>
<td>1447</td>
</tr>
<tr>
<td><code>uflow</code></td>
<td>1445</td>
</tr>
<tr>
<td><code>underflow</code></td>
<td>1445</td>
</tr>
<tr>
<td><code>basic_filebuf&lt;char&gt;</code></td>
<td>1441</td>
</tr>
<tr>
<td><code>basic_filebuf&lt;wchar_t&gt;</code></td>
<td>1441</td>
</tr>
<tr>
<td><code>basic_format_arg</code></td>
<td>755</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>756–757</td>
</tr>
<tr>
<td><code>handle</code></td>
<td>757</td>
</tr>
<tr>
<td><code>operator bool</code></td>
<td>757</td>
</tr>
<tr>
<td><code>sync</code></td>
<td>757</td>
</tr>
<tr>
<td><code>format</code></td>
<td>757</td>
</tr>
<tr>
<td><code>basic_format_args</code></td>
<td>758</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>758</td>
</tr>
<tr>
<td><code>get</code></td>
<td>758</td>
</tr>
<tr>
<td><code>basic_format_context</code></td>
<td>754</td>
</tr>
<tr>
<td><code>advance_to</code></td>
<td>755</td>
</tr>
<tr>
<td><code>char_type</code></td>
<td>754</td>
</tr>
<tr>
<td><code>formatter_type</code></td>
<td>754</td>
</tr>
<tr>
<td><code>iterator</code></td>
<td>754</td>
</tr>
<tr>
<td><code>locale</code></td>
<td>754</td>
</tr>
<tr>
<td><code>out</code></td>
<td>754</td>
</tr>
<tr>
<td><code>basic_format_parse_context</code></td>
<td>753</td>
</tr>
<tr>
<td><code>advance_to</code></td>
<td>753</td>
</tr>
<tr>
<td><code>begin</code></td>
<td>753</td>
</tr>
<tr>
<td><code>char_type</code></td>
<td>753</td>
</tr>
<tr>
<td><code>check_arg_id</code></td>
<td>754</td>
</tr>
<tr>
<td><code>const_iterator</code></td>
<td>753</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>753</td>
</tr>
<tr>
<td><code>end</code></td>
<td>753</td>
</tr>
<tr>
<td><code>iterator</code></td>
<td>753</td>
</tr>
<tr>
<td><code>next_arg_id</code></td>
<td>754</td>
</tr>
<tr>
<td><code>basic_fstream</code></td>
<td>1376, 1451</td>
</tr>
<tr>
<td><code>close</code></td>
<td>1453</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>1452</td>
</tr>
<tr>
<td><code>is_open</code></td>
<td>1453</td>
</tr>
<tr>
<td><code>open</code></td>
<td>1453</td>
</tr>
<tr>
<td><code>rdbuf</code></td>
<td>1452</td>
</tr>
<tr>
<td><code>swap</code></td>
<td>1452</td>
</tr>
</tbody>
</table>
Index of library names 1778

basic_fstream<char>, 1441
basic_fstream<wchar_t>, 1441
basic_ifstream, 1376, 1447
close, 1449
destructor, 1448
is_open, 1449
open, 1449
rdbuf, 1449
swap, 1448
basic_ifstream<char>, 1441
basic_ifstream<wchar_t>, 1441
basic_ios, 1376, 1388
bad, 1392
clear, 1392
destructor, 1389, 1390
copyfmt, 1391
destructor, 1390
eof, 1392
exceptions, 1392
fail, 1392
fill, 1391
good, 1392
imbue, 1390
init, 1390, 1406
move, 1391
narrow, 1390
operator bool, 1392
operator!, 1392
rdbuf, 1390
rdstate, 1392
set_rdbuf, 1392
setstate, 1392
swap, 1392
tie, 1390
widen, 1391
basic_ios<wchar_t>, 1380
basic_istream, 1414
destructor, 1414
operator=, 1414
swap, 1414
basic_istream<char>, 1376, 1404
destructor, 1406
gcount, 1410
get, 1410, 1411
getline, 1411
ignore, 1412
operator=, 1406
operator>>, 1407–1409, 1413
peek, 1412
putback, 1412
read, 1412
readsome, 1412
seekg, 1413
sentry, 1406
swap, 1406
sync, 1413
tellg, 1413
unget, 1412
basic_istream::sentry, 1406
destructor, 1407
operator bool, 1407
basic_istream<char>, 1403
basic_istream<wchar_t>, 1403
basic_istream_view, 1006
destructor, 1007
end, 1007
basic_istream_view::iterator, 1007
destructor, 1007
operator*, 1007
operator++, 1007
operator==, 1008
basic_istringstream, 1376, 1433
destructor, 1434, 1435
rdbuf, 1435
str, 1435, 1436
swap, 1435
view, 1435
basic_istringstream<char>, 1427
basic_istringstream<wchar_t>, 1427
basic_ostream, 1376, 1449
close, 1451
destructor, 1450
is_open, 1450
open, 1450, 1451
rdbuf, 1450
swap, 1450
basic_ostream<char>, 1417
destructor, 1417
flush, 1422
init, 1417
operator<<, 1419–1422
operator=, 1417
put, 1421
seekp, 1418
sentry, 1417
swap, 1417
tellp, 1418
write, 1421
basic_ostream::sentry, 1417
destructor, 1418
operator bool, 1418
basic_ostream<wchar_t>, 1417
basic_ostringstream, 1376, 1436
close, 1437, 1438
rdbuf, 1438
str, 1438
swap, 1438
view, 1438
basic_ostringstream<char>, 1427
Index of library names

basic_ostringstream<wchar_t>, 1427
basic_osyncstream, 1376, 1456
  constructor, 1457
set_emit_on_sync, 1457
basic_regex, 1508, 1517, 1536
  assign, 1520
  constructor, 1519
  flag_type, 1520
getloc, 1521
imbue, 1521
mark_count, 1520
operator=, 1520
swap, 1521
basic_streambuf, 1376, 1396
  constructor, 1397
destructor, 1398
eback, 1399
egptr, 1399
egptr, 1400
gbump, 1399
getloc, 1398
gptr, 1399
imbue, 1400
in_avail, 1398
operator=, 1399
overflow, 1402
pbackfail, 1402
pbase, 1400
pbump, 1400
pptr, 1400
pubimbue, 1398
pubseekoff, 1398
pubseekpos, 1398
pubsetbuf, 1398
pubsync, 1398
sbumpc, 1398
seekoff, 1400
seekpos, 1400
setbuf, 1400, 1433
setg, 1399
setp, 1400
sgetc, 1398
sgetn, 1399
showmanyc, 1401, 1445
snxtc, 1398
sputbackc, 1399
sputc, 1399
sputn, 1399
sungetc, 1399
swap, 1399
sync, 1400
uflow, 1401
underflow, 1401
xsgetn, 1401
xsputn, 1402
basic_streambuf<char>, 1395
basic_streambuf<wchar_t>, 1395
basic_string, 768, 786, 1427, 1696
  allocator_type, 768
append, 778
assign, 779
at, 777
back, 777
begin, 775
c_str, 783
capacity, 776
cbegin, 775
cend, 776
clear, 777
compare, 785, 786
const_iterator, 768
const_pointer, 768
const_reference, 768
const_reverse_iterator, 768
constructor, 773, 774
copy, 783
crbegin, 776
crend, 776
data, 783, 784
difference_type, 768
empty, 777
dern, 776
ends_with, 786
erase, 781, 789
erase_if, 789
find, 784
find_first_not_of, 784
find_first_of, 784
find_last_not_of, 784
find_last_of, 784
front, 777
get_allocator, 784
ggetline, 788, 789
insert, 780, 781
iterator, 768
length, 776
max_size, 776
operator basic_string_view, 784
operator*, 786, 787
operator=, 777, 778
operator<<, 788
operator>, 775
operator>>, 788
operator[], 777
pointer, 768
pop_back, 781
push_back, 778
rbegin, 776
reference, 768
rend, 776
replace, 781–783
reserve, 776, 1696
resize, 776
reverse_iterator, 768
rfind, 784
shrink_to_fit, 776
size, 776
size_type, 768
Index of library names 1780
tzdb_list, 1315
unordered associative containers, 838
valarray, 1228
begin(C&), 979
begin(initializer_list<E>), 538
begin(T (&)[N]), 979
bernoulli_distribution, 1195
   constructor, 1195
   p, 1195
   result_type, 1195
beta, 1239
   gamma_distribution, 1199
betal, 1239
betaf, 1239
bidirectional_iterator, 940
bidirectional_iterator_tag, 949
bidirectional_range, 992
big
   endian, 1173
binary_function
   zombie, 498
binary_negate
   zombie, 498
binary_search, 1122
bind, 700–701
bind1st
   zombie, 498
bind2nd
   zombie, 498
bind_front, 699
binder1st
   zombie, 498
binder2nd
   zombie, 498
binomial_distribution, 1195
   constructor, 1196
   p, 1196
   result_type, 1195
   t, 1196
bit_and, 697
   operator(), 697
bit_and<>, 697
   operator(), 697
bit_cast, 1171
bit_set, 1171
bit_clear, 1171
bit_reset, 1171
bit_xor, 698
   operator(), 698
bit_xor<>, 698
   operator(), 698
bitset, 628
   all, 632
   any, 632
   constructor, 630
   count, 632
   flip, 631
   none, 632
   operator<<, 632, 633
   operator<<=, 631
   operator==, 632
   operator>>, 632, 633
   operator>>=, 631
   operator[], 632, 633
   operator&, 633
   operator&,=, 630
   operator^, 633
   operator^=, 631
   operator~., 631
   operator|., 633
   operator|,=, 630
   reset, 631
   set, 631
   size, 632
   test, 632
   to_string, 632
   to_ullong, 632
   to_ulong, 632
   bool_constant, 715
   boolalpha, 1392
   borrowed_range, 990
   boyer_moore_horspool_searcher, 706
      constructor, 707
      operator(), 707
   boyer_moore_searcher, 706
      constructor, 706
      operator(), 706
   bsearch, 507, 1160
   btowc, 802
bucket
   unordered associative containers, 838
   bucket_count
   unordered associative containers, 837
   bucket_size
   unordered associative containers, 838
BUFSIZ, 1503
byte, 506
   operator<<, 509
   operator<<=, 509
   operator>>, 509
   operator>>=, 509
   operator&., 509
   operator^., 510
   operator~., 510
   operator|., 509
   operator|,=, 509
   to_integer, 510
   byte_string
wstring_convert, 1699

C

c16rtomb, 803
c32rtomb, 803
c8rtomb, 803, 805
c_encoding
  weekday, 1289
c_str
  basic_string, 783
  path, 1472
cacos
  complex, 1168
cacosh
  complex, 1169
call_once, 1603
calloc, 507, 649, 1680
canonical, 1491
capacity
  basic_string, 776
  vector, 864
casin
  complex, 1168
casinh
  complex, 1169
catan
  complex, 1168
catanh
  complex, 1169
category
  error_code, 576
  error_condition, 577
  locale, 1338
cauchy_distribution, 1203
    a, 1204
    b, 1204
    constructor, 1204
    result_type, 1203
cbefore_begin
  forward_list, 852
cbegin, 987
    basic_string, 775
    basic_string_view, 795
    tzdb_list, 1315
  unordered associative containers, 838
cbegin(const Ck), 979
cbrt, 1229
cbrtf, 1229
cbrtl, 1229
cdata, 990
ceil, 1229
  duration, 1265
  time_point, 1270
cceil, 1229
cceil, 1229
cend, 987
    basic_string, 776
    basic_string_view, 795
tzdb_list, 1315
  unordered associative containers, 838
cend(const Ck), 979
cerr, 1379
CHAR_BIT, 519
char_class_type
  regex_traits, 1515
CHAR_MAX, 519
CHAR_MIN, 519
char_traits, 762–764
  char_type, 762
  int_type, 762
  state_type, 762
char_type
  basic_format_context, 754
  basic_format_parse_context, 753
  char_traits, 762
chars_format, 739
  fixed, 739
  general, 739
  hex, 739
  scientific, 739
check_arg_id
  basic_format_parse_context, 754
chi_squared_distribution, 1203
  constructor, 1203
  n, 1203
  result_type, 1203
choose, 1245
  earliest, 1245
  latest, 1245
chrono, 1245
cin, 1379
clamp, 1138
classic
  locale, 1342
classic_table
cctype<
clear
  atomic_flag, 1568
  basic_ios, 1392
  basic_string, 777
  error_code, 576
  error_condition, 577
  forward_list, 853
  path, 1471
  unordered associative containers, 837
clearerr, 1503
clock, 1334
clock_cast, 1281
clock_t, 1334
clock_time_conversion, 1279
  operator(), 1279–1281
CLOCKS_PER_SEC, 1334
clog, 1379
close
  basic_filebuf, 1444
  basic_fstream, 1453

Index of library names
Index of library names
Index of library names

asinh, 1169
atan, 1168
atanh, 1169
cacos, 1168
cacosh, 1169
casin, 1168
casinh, 1169
catans, 1168
catanh, 1169
cos, 1169
cosh, 1169
exp, 1169
imag, 1166, 1168
log, 1169
log10, 1169
norm, 1168
operator"i", 1170
operator"if", 1170
operator"il", 1170
operator*, 1167
operator=, 1166, 1167
operator+, 1167
operator=, 1166
operator-, 1167
operator=, 1166
operator/, 1167
operator/=, 1166, 1167
operator<, 1168
operator==, 1167
operator>>, 1167
polar, 1168
pow, 1169
proj, 1168
real, 1166, 1168
sin, 1169
sinf, 1169
sqrt, 1169
tan, 1169
tanh, 1169
value_type, 1164
concat
path, 1471
condition_variable, 1604
constructor, 1605
destructor, 1605
notify_all, 1605
notify_one, 1605
wait, 1605, 1606
wait_for, 1606, 1607
wait_until, 1606, 1607
condition_variable_any, 1608
constructor, 1608
destructor, 1608
notify_all, 1609
notify_one, 1608
wait, 1609
wait_for, 1609, 1610

wait_until, 1609, 1610
conditional_t, 712
conj, 1170
complex, 1168
conjunction, 731
conjunction_v, 715
const_iterator
basic_format_parse_context, 753
basic_string, 768
basic_string_view, 792, 795
const_local_iterator
unordered associative containers, 830
const_mem_fun1_ref_t
zombie, 498
const_mem_fun1_t
zombie, 499
const_mem_fun_ref_t
zombie, 499
const_mem_fun_t
zombie, 499
const_pointer
allocator_traits, 647
basic_string, 768
basic_string_view, 792
scoped_allocator_adaptor, 681
const_pointer_cast
shared_ptr, 668
const_reference
basic_string, 768
basic_string_view, 792
const_reverse_iterator
basic_string, 768
basic_string_view, 792
const_void_pointer
allocator_traits, 647
scoped_allocator_adaptor, 681
construct
allocator_traits, 647
polymorphic_allocator, 676
scoped_allocator_adaptor, 684
construct_at, 1159
constructible_from, 560
consume
memory_order, 1543
contains
ordered associative containers, 826
unordered associative containers, 837
contiguous_iterator, 941
contiguous_iterator_tag, 949
contiguous_range, 992
converted
wstring_convert, 1699
convertible_to, 557
copy, 1099
basic_string, 783
basic_string_view, 796
path, 1491
copy_backward, 1100
copy_constructible, 561
copy_file, 1493
copy_if, 1100
copy_n, 1099
copy_options, 1479
copy_symlink, 1493
copyable, 564
copyfmt
  basic_ios, 1391
copysign, 1229
copysignf, 1229
copysignl, 1229
coroutine_handle, 547
  address, 548
  constructor, 548
done, 548
from_address, 548
from_promise, 548
hash, 549
operator bool, 548
operator!=, 549
operator(), 548
operator<=>, 549
operator=, 548
operator==, 549
promise, 548
resume, 548
coroutine_handle<noop_coroutine_promise>, 549
  address, 550
done, 549
operator bool, 549
operator=, 549
promise, 550
resume, 549
cos, 1229
  complex, 1169
  valarray, 1221
cosf, 1229
cosh, 1229
  complex, 1169
  valarray, 1221
coshf, 1229
coshl, 1229
cosl, 1229
count, 1093
  bitset, 632
  counted_iterator, 970
duration, 1262
  ordered associative containers, 826
  unordered associative containers, 837
count_down
  latch, 1613
count_if, 1093
counted_iterator, 968
  base, 970
  constructor, 970
count, 970
iter_move, 972
iter_swap, 972
operator*, 970
operator*, 971
operator++, 971
operator==, 971
operator=, 972
operator==, 972
operator<=>, 972
operator=, 972
operator[](), 970
operator=, 971
counting_semaphore
  acquire, 1612
  constructor, 1612
  max, 1611
  release, 1612
  try_acquire, 1612
  try_acquire_for, 1612
  try_acquire_until, 1612
count_one, 1172
count_zero, 1172
countr_one, 1173
countr_zero, 1172
cout, 1379
crbegin, 988
  basic_string, 776
  basic_string_view, 795
crbegin(const C& c), 979
create_directories, 1494
create_directory, 1494
create_directory_symlink, 1494
create_hard_link, 1495
create_symlink, 1495
cref
  reference_wrapper, 690
crend, 988
  basic_string, 776
  basic_string_view, 795
crend(const C& c), 979
cshift
  valarray, 1219
cctime, 1334
cctype, 1343
do_is, 1344
do_narrow, 1345
do_scan_not, 1344
do_tolower, 1345
do_toupper, 1345
do_widen, 1345
is, 1344
narrow, 1344
scan_is, 1344
scan_not, 1344
tolower, 1344
toupper, 1344
widen, 1344
cctype<char>, 1346
Index of library names 1785
Index of library names

D

dangling, 998
data, 989
  array, 844
  basic_string, 783, 784
  basic_string_view, 796
  single_view, 1000
  span, 920
  vector, 865
data(C & c), 980
data(initializer_list< E >), 980
data(T (&array)[N]), 980
date
  leap_second, 1325
  date_order
  time_get, 1363
day, 1282
  constructor, 1282
  from_stream, 1283

month_day, 1292
ok, 1283
operator unsigned, 1283
operator "d", 1283
operator+, 1283
operator++, 1282
operator+=, 1283
operator-, 1283
operator--, 1282
operator-=, 1283
operator<<, 1283
operator<<, 1283
operator==, 1283
year_month_day, 1299
year_month_day_last, 1302
days, 1245
DBL_DECIMAL_DIG, 520
DBL_DIG, 520
DBL_EPSILON, 520
DBL_HAS_SUBNORM, 520
DBL_MANT_DIG, 520
DBL_MAX, 520
DBL_MAX_10_EXP, 520
DBL_MAX_EXP, 520
DBL_MIN, 520
DBL_MIN_10_EXP, 520
DBL_MIN_EXP, 520
DBL_TRUE_MIN, 520
dallocate
  allocator, 648
  allocator_traits, 647
  memory_resource, 673
  polymorphic_allocator, 675
  scoped_allocator_adaptor, 684
dallocate_bytes
  polymorphic_allocator, 675
dallocate_object
  polymorphic_allocator, 675
dec, 1394, 1420
decay, 728
decay-copy, 482
decay_t, 712
DECIMAL_DIG, 520
decimal_point
  moneypunct, 1371
  numpunct, 1360
decclare_no_pointers, 642
decclare_reachable, 642
dclval, 584
default_delete
  constructor, 650
  operator(), 650
default_error_condition
  error_category, 573, 574
  error_code, 576
default_initializable, 561
default_random_engine, 1190
default_searcher, 705
  constructor, 705
operator(), 705
default_sentinel, 927
default_sentinel_t, 968
default_zone
  zoned_traits<\text{const time_zone}>, 1320
defaultfloat, 1394
defer_lock, 1594
defer_lock_t, 1594
delete
  operator, 500, 501, 524–527, 649
derenum_absent, 513
derenum_indeterminate, 513
derenum_min
  numeric_limits, 517
derenum_present, 513
densities
  piecewise_constant_distribution, 1208
  piecewise_linear_distribution, 1210
depth
  recursive_directory_iterator, 1490
ddeque, 845
  constructor, 847, 848
  emplace, 848
  erase, 849
  erase_if, 849
  insert, 848
  push_back, 848
  push_front, 848
  resize, 848
  shrink_to_fit, 848
derived_from, 556
destroy, 1159
  allocator_traits, 648
  coroutine_handle, 548
  coroutine_handle<noop_coroutine_-
    promise>, 549
  polymorphic_allocator, 676
  scoped_allocator_adaptor, 684
destroy_at, 1159
destroy_n, 1159, 1160
destroying_delete, 523
destroying_delete_t, 523
destructible, 560
detach
  jthread, 1584
  thread, 1582
difference_type
  allocator, 648
  allocator_traits, 647
  basic_string, 768
  basic_string_view, 792
  pointer_traits, 641
  scoped_allocator_adaptor, 681
difftime, 1334
digits
  numeric_limits, 515
digits10
  numeric_limits, 515
directory_entry, 1482
  assign, 1484
  constructor, 1484
  exists, 1484
  file_size, 1485
  is_block_file, 1484
  is_character_file, 1484
  is_directory, 1485
  is_fifo, 1485
  is_other, 1485
  is_regular_file, 1485
  is_socket, 1485
  is_symlink, 1485
  last_write_time, 1485
  operator const filesystem::path&, 1484
  operator<\text{\textgreater{}}, 1486
  operator\text{\textless{}=}, 1486
  path, 1484
  refresh, 1484
  replace_filename, 1484
  status, 1485
  symlink_status, 1486
  directory_iterator, 1486
  begin, 1487
  constructor, 1487
  end, 1488
  increment, 1487
  operator++, 1487
  operator=, 1487
directory_options, 1481
disable_recursion_pending
  recursive_directory_iterator, 1490
disable_sized_range, 991
disable_sized_sentinel_for, 939
discard_block_engine, 1186, 1187
  constructor, 1187
  result_type, 1187
discrete_distribution, 1205
  constructor, 1206
  probabilities, 1206
  result_type, 1205
disjunction, 732
disjunction_v, 715
distance, 950, 952
div, 507, 1504
div_t, 507
divides, 691
  operator(), 691
  divides<\text{\textgreater{}}, 691
  operator(), 691
do_allocate
  memory_resource, 673
  monotonic_buffer_resource, 681
  synchronized_pool_resource, 679
  unsynchronized_pool_resource, 679
do_always_noconv
  codecvt, 1351
do_close
message, 1373
do_compare
    collate, 1361
do_curr_symbol
        moneypunct, 1372
do_date_order
        time_get, 1364
do_deallocate
        memory_resource, 673
        monotonic_buffer_resource, 681
        synchronized_pool_resource, 679
        unsynchronized_pool_resource, 679
do_decimal_point
        moneypunct, 1371
        numpunct, 1360
do_encoding
        codecvt, 1351
do_falsename
        numpunct, 1360
do_frac_digits
        moneypunct, 1372
do_get
        messages, 1373
        money_get, 1368
        num_get, 1353, 1355
        time_get, 1365
do_get_date
        time_get, 1364
do_get_monthname
        time_get, 1365
do_get_time
        time_get, 1364
do_get_weekday
        time_get, 1365
do_get_year
        time_get, 1365
do_grouping
        moneypunct, 1372
        numpunct, 1360
do_hash
        collate, 1362
do_in
        codecvt, 1349
do_is
        ctype, 1344
do_is_equal
        memory_resource, 674
        monotonic_buffer_resource, 681
        synchronized_pool_resource, 679
        unsynchronized_pool_resource, 679
do_length
        codecvt, 1351
do_max_length
        codecvt, 1351
do_narrow, 1347
        ctype, 1345
        ctype<char>, 1347
do_neg_format
        moneypunct, 1372
do_negative_sign
        moneypunct, 1372
do_open
        messages, 1373
do_out
        codecvt, 1349
do_pos_format
        moneypunct, 1372
do_positive_sign
        moneypunct, 1372
do_put
        money_put, 1369
        num_put, 1356, 1358
        time_put, 1366
do_scan_is
        ctype_base, 1344
do_scan_not
        ctype, 1344
do_thousands_sep
        moneypunct, 1371
        numpunct, 1360
do_tolower
        ctype, 1345
        ctype<char>, 1347
do_toupper
        ctype, 1345
        ctype<char>, 1347
do_transform
        collate, 1361
do_truename
        numpunct, 1360
do_unshift
        codecvt, 1350
do_widen
        ctype, 1347
        ctype<char>, 1347
domain_error, 566, 567
        constructor, 567
done
        coroutine_handle, 548
        coroutine_handle<noop_coroutine_<promise>>, 549
double_t, 1229
drop_view, 1023
        base, 1023
        begin, 1024
        constructor, 1024
        end, 1023
        size, 1023
drop_while_view, 1024
        base, 1024
        begin, 1025
        constructor, 1025
        end, 1024
        pred, 1025
duration, 1261
        abs, 1266
        ceil, 1265
constructor, 1262
count, 1262
duration_cast, 1265
floor, 1265
from_stream, 1267
max, 1263
min, 1263
operator "h, 1266
operator "min, 1266
operator "ms, 1266
operator "ns, 1266
operator "s, 1266
operator "us, 1266
operator *, 1264
operator**, 1263
operator+, 1262, 1269
operator++, 1263
operator+, 1263
operator-, 1262, 1269
operator-, 1263
operator/-, 1263
operator<, 1265
operator<<, 1267
operator<=, 1265
operator==, 1264
operator>, 1265
operator>>, 1265
operator%, 1264
operator%?, 1263
round, 1266
zero, 1263
duration_cast, 1265
duration, 1265
duration_values, 1260
max, 1260
min, 1260
zero, 1260
dynamic_extent, 915
dynamic_pointer_cast
shared_ptr, 667

E
EB2BIG, 569
EACCESS, 569
EADDRINUSE, 569
EADDRNOTAVAIL, 569
EAFNOSUPPORT, 569
EAGAIN, 569
EALREADY, 569
earliest
choose, 1245
eback
basic_streambuf, 1399
EBADF, 569
EBADMSG, 569
EBUSY, 569
ec
from_chars_result, 739
to_chars_result, 739
ECANCELED, 569
ECHILD, 569
ECMAScript
syntax_option_type, 1512
ECONNABORTED, 569
ECONNREFUSED, 569
ECONNRESET, 569
EDEADLK, 569
EDESTADDRREQ, 569
EDOM, 569
EXIST, 569
EFAULT, 569
EFBIG, 569
egptr
basic_streambuf, 1399
egrep
syntax_option_type, 1512
EHOSTUNREACH, 569
EIDRM, 569
EILSEQ, 569
EINPROGRESS, 569
EINVAL, 569
EIO, 569
EISCONN, 569
EISDIR, 569
elements_view, 1039
base, 1039
begin, 1039
constructor, 1040
end, 1039
size, 1039
elements_view::iterator, 1040
base, 1041
constructor, 1041
operator+, 1042
operator++, 1041
operator+=, 1042
operator-, 1042, 1043
operator--, 1042
operator<, 1042
operator<=, 1042
operator<>, 1042
operator<=>, 1042
operator==, 1042
operator>, 1042
operator>=, 1042
operator%?, 1043
elements_view::sentinel, 1043
base, 1043
constructor, 1043
operator-, 1043
operator==, 1043
ellint_1, 1241
ellint_1f, 1241
ellint_1l, 1241
ellint_2, 1241
ellint_2f, 1241
ellint_2l, 1241
ellint_3, 1241
ellint_3f, 1241
ellint_3l, 1241
ELOOP, 569
EMFILE, 569
emit
\quad basic_syncbuf, 1455
\quad emit_on_flush, 1422
EMLINK, 569
emplace
\quad any, 626
\quad deque, 848
\quad optional, 607
\quad ordered associative containers, 822
\quad priority_queue, 913
\quad unordered associative containers, 833
\quad variant, 618, 619
emplace_after
\quad forward_list, 853
emplace_front
\quad forward_list, 852
emplace_hint
\quad ordered associative containers, 822
\quad unordered associative containers, 833
empty, 989
\quad basic_string, 777
\quad basic_string_view, 795
\quad match_results, 1525
\quad path, 1475
\quad span, 920
\quad subrange, 997
empty(C& c), 980
empty(initializer_list\langle E\rangle), 980
empty(T (&array)[N]), 980
empty_view, 999
EMSGSIZE, 569
enable_borrowed_range, 991
enable_if, 728
enable_if_t, 712
enable_shared_from_this, 671
\quad constructor, 672
\quad operator=, 672
\quad shared_from_this, 672
\quad weak_from_this, 672
enable_view, 992
ENAMETOOLONG, 569
encoding
\quad codecvt, 1349
end, 537, 986
\quad array, 843
\quad basic_format_parse_context, 753
\quad basic_istream_view, 1007
\quad basic_string, 776
\quad basic_string_view, 795
common_view, 1036
directory_iterator, 1488
drop_view, 1023
drop_while_view, 1024
elements_view, 1039
filter_view, 1010
initializer_list, 538
iota_view, 1002
join_view, 1025
match_results, 1526
path, 1477
recursive_directory_iterator, 1490
reverse_view, 1038
single_view, 1000
span, 921
split_view, 1030
split_view::outer_iterator::value_type, 1034
subrange, 997
sys_info, 1317
take_view, 1019
take_while_view, 1021
transform_view, 1014, 1015
tzdb_list, 1315
unordered associative containers, 838
valarray, 1228
end(C&), 979
end(initializer_list\langle E\rangle), 538
end(T (&array)[N]), 979
endian, 1173
\quad big, 1173
\quad little, 1173
\quad native, 1173
endl, 1420, 1422
ends, 1422
ends_with
\quad basic_string, 786
\quad basic_string_view, 797
ENETDOWN, 569
ENETRESET, 569
ENETUNREACH, 569
ENFILE, 569
ENOBUFFS, 569
ENODATA, 569
ENODEV, 569
ENOENT, 569
ENODEV, 569
ENOEXEC, 569
ENOLCK, 569
ENOLINK, 569
ENOMEM, 569
ENOPROTOOPT, 569
ENOSPC, 569
ENOSR, 569
ENOSTR, 569
ENOSYS, 569
ENOTCONN, 569
ENOTDIR, 569

Index of library names 1790
Index of library names
ETXTBSY, 569
EWOULDBLOCK, 569

exception, 534
constructor, 534
destructor, 534
operator=, 534
what, 534

exception_ptr, 535

exceptions
basic_ios, 1392

exchange, 583
atomic, 1554
atomic<floating-point>, 1554
atomic<integral>, 1554
atomic<shared_ptr<T>>, 1564
atomic<T>, 1554
atomic<weak_ptr<T>>, 1566
atomic_ref, 1548
atomic_ref<floating-point>, 1548
atomic_ref<integral>, 1548
atomic_ref<T*>, 1548

exclusive_scan, 1148, 1149

EXDEV, 569

execution
par, 738
par_unseq, 738
seq, 738

execution::parallel_policy, 738

execution::parallel_unsequenced_policy, 738

execution::sequenced_policy, 738

execution::unsequenced_policy, 738

exists, 1496
directory_entry, 1484

exit, 87, 89, 160, 487, 507, 522, 528
EXIT_FAILURE, 507
EXIT_SUCCESS, 507

exp, 1229
cmplx, 1169
valarray, 1221

exp2, 1229
exp2f, 1229
exp2l, 1229
expf, 1229
expint, 1242
expintf, 1242
expintl, 1242

expired
weak_ptr, 670

expl, 1229

exp, 1229

exp1, 1229
exp1f, 1229
exp1l, 1229

exponential_distribution, 1198
constructor, 1199
lambda, 1199
result_type, 1198

extended
syntax_option_type, 1512

extension
path, 1474

extent, 723
extent_v, 714

extract
ordered associative containers, 825
unordered associative containers, 836

extreme_value_distribution, 1200
a, 1201
b, 1201
constructor, 1201
result_type, 1200

F
fabs, 1229
fabsf, 1229
fabsl, 1229

facet
locale, 1339

fail
basic_ios, 1392

failed
ostreambuf_iterator, 978

failure
ios_base, 1383
false_type, 715

false
numpunct, 1360

fclose, 1444, 1503
fdim, 1229
fdimf, 1229
fdiml, 1229

FE_ALL_EXCEPT, 1161
FE_DFL_ENV, 1161
FE_DIVBYZERO, 1161
FE_DOWNWARD, 1161
FE_INEXACT, 1161
FE_INVALID, 1161
FE_OVERFLOW, 1161
FE_TONEAREST, 1161
FE_TOWARDZERO, 1161
FE_UNDERFLOW, 1161
FE_UPWARD, 1161

fenv_t, 1161

fenv_t, 1161

fesetenv, 1161
fesetexceptflag, 1161
fesetround, 1161

fenv_t, 1161
feof, 1503

ferror, 1503

fesetenv, 1161
fesetexceptflag, 1161
fesetround, 1161

fetch_add
atomic<floating-point>, 1560
atomic<integral>, 1558
atomic<T*>, 1561
atomic_ref<floating-point>, 1551
atomic_ref<integral>, 1550
atomic_ref<T*>, 1552
fetch_and
atomic<integral>, 1558
atomic_ref<integral>, 1550
fetch_or
atomic<integral>, 1558
atomic_ref<integral>, 1550
fetch_sub
atomic<floating-point>, 1560
atomic<integral>, 1558
atomic<T*>, 1561
atomic_ref<floating-point>, 1551
atomic_ref<integral>, 1550
atomic_ref<T*>, 1552
fetch_xor
atomic<integral>, 1558
atomic_ref<integral>, 1550
fetestexcept, 1161
feupdateenv, 1161
fexcept_t, 1161
fflush, 1503
fgetc, 1503
fgetpos, 1503
fgets, 1503
fgetwc, 802
fgetws, 802
FILE, 1503
file_clock, 1277
now, 1277
file_size, 1496
directory_entry, 1485
file_status, 1481
constructor, 1482
permissions, 1482
type, 1482
file_time, 1245
from_stream, 1278
operator<<, 1277
file_type, 1479
filebuf, 1376, 1441
filename
path, 1474
FILENAME_MAX, 1503
filesystem_error, 1478
constructor, 1478
path1, 1478
path2, 1478
what, 1479
fill, 1106
array, 844
basic_ios, 1391
fill_n, 1106
filter_view, 1010
base, 1010
begin, 1011
constructor, 1011
end, 1010
iterator, 1011
pred, 1011
sentinel, 1013
filter_view::iterator
base, 1012
constructor, 1012
iter_move, 1013
iter_swap, 1013
operator*, 1012
operator++, 1012
operator->, 1012
operator=, 1012
operator==, 1013
filter_view::sentinel
base, 1013
constructor, 1013
operator==, 1013
find, 1089
basic_string, 784
basic_string_view, 798
ordered associative containers, 826
unordered associative containers, 837
find_end, 1090
find_first_not_of
basic_string, 784
basic_string_view, 798
find_first_of, 1091
basic_string, 784
basic_string_view, 798
find_if, 1089
find_if_not, 1089
find_last_not_of
basic_string, 784
basic_string_view, 799
find_last_of
basic_string, 784
basic_string_view, 798
first
local_info, 1318
span, 919, 920
first_argument_type
zombie, 499
fisher_distribution
result_type, 1204
fisher_f_distribution, 1204
constructor, 1204
m, 1205
n, 1205
fixed, 1394
chars_format, 739
flag_type
basic_regex, 1520
flags
ios_base, 1342, 1385
flip
bitset, 631
vector<bool>, 868
float_denorm_style, 512, 513
numeric_limits, 516
float_round_style, 512, 513
float_t, 1229
floating_point, 558
floor, 1229
duration, 1265
time_point, 1270
floorf, 1229
floorl, 1229
FLT_DECIMAL_DIG, 520
FLT_DIG, 520
FLT_EPSILON, 520
FLT_EVAL_METHOD, 520
FLT_HAS_SUBNORM, 520
FLT_MANT_DIG, 520
FLT_MAX, 520
FLT_MAX_10_EXP, 520
FLT_MAX_EXP, 520
FLT_MIN, 520
FLT_MIN_10_EXP, 520
FLT_MIN_EXP, 520
FLT_RADIX, 520
FLT_ROUNDS, 520
FLT_TRUE_MIN, 520
flush, 1385, 1406, 1418, 1422
basic ostream, 1422
flush emit, 1422
fma, 1229
fmaf, 1229
fmal, 1229
fmax, 1229
fmaxf, 1229
fmaxl, 1229
fmin, 1229
fminf, 1229
fminl, 1229
fmod, 1229
fmodf, 1229
fmodl, 1229
fmtflags
ios_base, 1383, 1423
fopen, 1444, 1503
FOPEN_MAX, 1503
for each, 1088
for_each, 1089
format, 749, 1326–1330
basic_format_arg: handle, 757
formatter<chrono::zoned_time>, 1330
match_results, 1526
format_args, 741
format_args_t, 741, 742
format_context, 741, 754
format_default, 1513
format_error, 758
constructor, 758
format_first_only, 1513, 1530
format_no_copy, 1513, 1530
format_parse_context, 741
format_sed, 1513
format_to, 749
format_to_n, 750
format_to_n_result, 741
out, 741
size, 741
formatted_size, 750
formatter, 751
specializations
arithmetic types, 752
character types, 751
chrono::file_time, 1330
chrono::gps_time, 1329
chrono::local_time, 1330
chrono::local_time, 1329
crono::sys_time, 1329
crono::tai_time, 1329
crono::utc_time, 1329
crono::zoned_time, 1330
nullptr_t, 752
pointer types, 752
string types, 751
formatter<chrono::zoned_time>
format, 1330
formatter_type
basic_format_context, 754
forward, 583
forward_as_tuple, 596
tuple, 596
forward_iterator, 940
forward_iterator_tag, 949
forward_list
before_begin, 852
cbefore_begin, 852
clear, 853
constructor, 851, 852
emplace_after, 853
emplace_front, 852
erase, 855
erase_after, 853
erase_if, 855
front, 852
insert_after, 852, 853
merge, 855
pop, 852
push_front, 852
remove, 854
remove_if, 854
resize, 853
reverse, 855
sort, 855
splice_after, 854
unique, 854
forward_range, 992
FP_FAST_FMA, 1229
FP_FAST_FMAF, 1229
FP_FAST_FMAL, 1229
FP_ILOGBO, 1229
FP_ILOGBNAN, 1229
FP_INFINITE, 1229

Index of library names 1794
Index of library names

FP_NAN, 1229
FP_NORMAL, 1229
FP_SUBNORMAL, 1229
FP_ZERO, 1229
fpclassify, 1229
fpos, 1376, 1380, 1387, 1388
   state, 1388
fpos_t, 1503
fprintf, 1376, 1380, 1387, 1388
state, 1388
fputc, 1503
fputs, 1503
fputwc, 802
fputws, 802
frac_digits
moneypunct, 1371
fread, 1503
free, 507, 649
freeze
   ostrstream, 1691
   strstream, 1692
   strstreambuf, 1687
freopen, 1503
frexp, 1229
frexpf, 1229
frexpl, 1229
from_address
   coroutine_handle, 548
from_bytes
   wstring_convert, 1699
from_chars, 741
from_chars_result, 739
   ec, 739
   ptr, 739
from_promise
   coroutine_handle, 548
from_stream
   day, 1283
   duration, 1267
   file_time, 1278
   gps_time, 1277
   local_time, 1279
   month, 1285
   month_day, 1293
   sys_time, 1272
   tai_time, 1275
   utc_time, 1274
   weekday, 1290
   year, 1288
   year_month, 1297
   year_month_day, 1300
from_sys
   utc_clock, 1273
from_time_t
   system_clock, 1271
from_utc
   gps_clock, 1276
   tai_clock, 1275
front
   basic_string, 777
   basic_string_view, 796
   forward_list, 852
   span, 920
tzdb_list, 1314
view_interface, 994
front_insert_iterator, 958
   constructor, 958
   operator*, 958
   operator++, 958
   operator=, 958
front_insertter, 958
fscanf, 1503
fseek, 1444, 1503
fsetpos, 1503
fstream, 1376, 1441
ftell, 1503
function, 702
   constructor, 703
destructor, 704
   invocation, 704
operator bool, 704
operator(), 704
operator=, 703, 704
operator==, 704
result_type, 702
swap, 704, 705
target, 704
target_type, 704
future, 1621
   constructor, 1622
get, 1622
   operator=, 1622
   share, 1622
   valid, 1623
   wait, 1623
   wait_for, 1623
   wait_until, 1623
future_category, 1617
future_errc, 1616
   make_error_code, 1617
   make_error_condition, 1617
future_error, 1617
   code, 1618
   constructor, 1617
   what, 1618
fwide, 802
fwprintf, 802
fwrite, 1503
fwscanf, 802
G
gamma_distribution, 1199
   alpha, 1199
   beta, 1199
   constructor, 1199
   result_type, 1199
gbump
   basic_streambuf, 1399
Index of library names

gcd, 1153
gcount
  basic_istream, 1410
general
  chars_format, 739
  GENERALIZED_NONCOMMUTATIVE_SUM, 1145
  GENERALIZED_SUM, 1145
generate, 1107
  seed_seq, 1192
generate_canonical, 1193
generate_n, 1107
generic_category, 573, 574
generic_string
  path, 1473
generic_u16string
  path, 1473
generic_u32string
  path, 1473
generic_u8string
  path, 1473
generic_wstring
  path, 1473
geometric_distribution, 1196
  constructor, 1197
  p, 1197
  result_type, 1196
get
  array, 845
  basic_format_args, 758
  basic_istream, 1410, 1411
  future, 1622
  messages, 1373
  money_get, 1368
  num_get, 1353
  pair, 589, 590
  reference_wrapper, 689
  shared_future, 1625
  shared_ptr, 663
  subrange, 998
  time_get, 1363
  tuple, 598
  unique_ptr, 654
  variant, 620
get_allocator
  basic_string, 784
  basic_stringbuf, 1431
  basic_syncbuf, 1455
  match_results, 1527
get_date
  time_get, 1363
get_default_resource, 677
get_deleter
  shared_ptr, 668
  unique_ptr, 654
get_future
  packaged_task, 1628
  promise, 1620
get_id
  jthread, 1584
good
  basic_ios, 1392
  this_thread, 1585
  thread, 1582
get_if, 620, 621
  variant, 620, 621
get_info
  time_zone, 1319
  zoned_time, 1324
get_leap_second_info, 1274
get_local_time
  zoned_time, 1323
get_money, 1424
get_monthname
  time_get, 1363
get_new_handler, 501, 529
get_pointer_safety, 643
get_stop_source
  jthread, 1585
get_stop_token
  jthread, 1585
get_sys_time
  zoned_time, 1324
get_temporary_buffer
  zombie, 499
get_terminate, 501, 535
get_time, 1425
  time_get, 1363
get_time_zone
  zoned_time, 1323
get_token
  stop_source_sc, 1578
get_tzdb, 1315
get_tzdb_list, 1315
get_unexpected
  zombie, 499
get_weekday
  time_get, 1363
get_wrapped
  basic_syncbuf, 1455
get_year
  time_get, 1363
getc, 1503
getenv, 507, 550
getenv
  basic_istream, 1410
  basic_string, 788, 789
getloc, 1517
getloc
  basic_regex, 1521
  basic_stringbuf, 1398
  ios_base, 1386
gets
  zombie, 499
getwc, 802
getwchar, 802
global
  locale, 1341
gmtime, 1334
good
  basic_ios, 1392
gps_clock, 1275
  from_utc, 1276
now, 1276
to_utc, 1276
gps_seconds, 1245
gps_time, 1245
  from_stream, 1277
operator<<, 1276
gptr
  basic_streambuf, 1399
greater, 693, 695
    operator(), 693
    partial_ordering, 539
    strong_ordering, 541
    weak_ordering, 540
greater>, 693
    operator(), 693
greater_equal, 693, 696
    operator(), 693
greater_equal>, 694
    operator(), 694
grep
  syntax_option_type, 1512
grouping
    moneypunct, 1371
    numpunct, 1360
gslice, 1224
    constructor, 1225
    size, 1225
    start, 1225
    stride, 1225
gslice_array, 1225
    operator*, 1226
    operator+=, 1226
    operator-=, 1226
    operator/=, 1226
    operator<<=, 1226
    operator%=, 1226
    operator^=, 1226
    operator|, 1226
    value_type, 1225

H
handle
  basic_format_arg, 757
hard_link_count, 1496
  directory_entry, 1485
hardware_concurrency
jthread, 1585
  thread, 1582
hardware_constructive_interference_size, 529
hardware_destructive_interference_size, 529
has_denorm_loss
numeric_limits, 516
has_extension
  path, 1475
has_facet
  locale, 1342
has_filename
  path, 1475
has_infinity
  numeric_limits, 516
has_parent_path
  path, 1475
has_quiet_NaN
  numeric_limits, 516
has_relative_path
  path, 1475
has_root_directory
  path, 1475
has_root_name
  path, 1475
has_root_path
  path, 1475
has_signaling_NaN
  numeric_limits, 516
has_single_bit, 1171
has_stem
  path, 1475
has_unique_object_representations, 722, 723
has_value
  any, 627
  optional, 608
has_virtual_destructor, 722
has_virtual_destructor_v, 714
hash, 707
  collate, 1361
  coroutine_handle, 549
  error_code, 578
  monostate, 623
  optional, 612
  pmr::string, 791
  pmr::u16string, 791
  pmr::u32string, 791
  pmr::wstring, 791
  shared_ptr, 672
  string, 791
  string_view, 800
  thread::id, 1580
  type_index, 737
  u16string, 791
  u16string_view, 800
  u32string, 791
  u32string_view, 800
  u8string_view, 800
  unique_ptr, 672
  variant, 623
  wstring, 791
  wstring_view, 800
hash_code, 633
  type_index, 736
  type_info, 530
<table>
<thead>
<tr>
<th>hash_function</th>
<th>unordered associative containers, 833</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_value</td>
<td>path, 1477</td>
</tr>
<tr>
<td>hasher</td>
<td>unordered associative containers, 830</td>
</tr>
<tr>
<td>hermite</td>
<td>1242</td>
</tr>
<tr>
<td>hermitef</td>
<td>1242</td>
</tr>
<tr>
<td>hermitel</td>
<td>1242</td>
</tr>
<tr>
<td>hex</td>
<td>1394</td>
</tr>
<tr>
<td>chars_format</td>
<td>739</td>
</tr>
<tr>
<td>hexfloat</td>
<td>1394</td>
</tr>
<tr>
<td>hh_mm_ss</td>
<td></td>
</tr>
<tr>
<td>hours</td>
<td>1312</td>
</tr>
<tr>
<td>is_negative</td>
<td>1312</td>
</tr>
<tr>
<td>minutes</td>
<td>1312</td>
</tr>
<tr>
<td>operator precision</td>
<td>1312</td>
</tr>
<tr>
<td>seconds</td>
<td>1312</td>
</tr>
<tr>
<td>subseconds</td>
<td>1312</td>
</tr>
<tr>
<td>to_duration</td>
<td>1312</td>
</tr>
<tr>
<td>high_resolution_clock</td>
<td>1278</td>
</tr>
<tr>
<td>hms</td>
<td>1311</td>
</tr>
<tr>
<td>holds_alternative</td>
<td>620</td>
</tr>
<tr>
<td>variant</td>
<td>620</td>
</tr>
<tr>
<td>hours</td>
<td>1245</td>
</tr>
<tr>
<td>hh_mm_ss</td>
<td>1312</td>
</tr>
<tr>
<td>HUGE_VAL</td>
<td>1229</td>
</tr>
<tr>
<td>HUGE_VALF</td>
<td>1229</td>
</tr>
<tr>
<td>HUGE_VALL</td>
<td>1229</td>
</tr>
<tr>
<td>hypot</td>
<td>1229</td>
</tr>
<tr>
<td>3-argument form, 1238</td>
<td></td>
</tr>
<tr>
<td>hypotf</td>
<td>1229</td>
</tr>
<tr>
<td>hypotl</td>
<td>1229</td>
</tr>
</tbody>
</table>

**I**

<table>
<thead>
<tr>
<th>icase</th>
<th>syntax_option_type, 1512</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>locale, 1340</td>
</tr>
<tr>
<td></td>
<td>thread, 1579</td>
</tr>
<tr>
<td>identity</td>
<td>698</td>
</tr>
<tr>
<td>ifstream</td>
<td>1376, 1441</td>
</tr>
<tr>
<td>ignore</td>
<td>596</td>
</tr>
<tr>
<td>basic_istream</td>
<td>1412</td>
</tr>
<tr>
<td>ilogb</td>
<td>1229</td>
</tr>
<tr>
<td>ilogbf</td>
<td>1229</td>
</tr>
<tr>
<td>ilogbl</td>
<td>1229</td>
</tr>
<tr>
<td>imag</td>
<td>1170</td>
</tr>
<tr>
<td>complex</td>
<td>1166, 1168</td>
</tr>
<tr>
<td>imaxabs</td>
<td>1504</td>
</tr>
<tr>
<td>imaxdiv</td>
<td>1504</td>
</tr>
<tr>
<td>imaxdiv_t</td>
<td>1504</td>
</tr>
<tr>
<td>imbed</td>
<td>1517</td>
</tr>
<tr>
<td>basic_filebuf</td>
<td>1447</td>
</tr>
<tr>
<td>basic_ios</td>
<td>1390</td>
</tr>
<tr>
<td>basic_regex</td>
<td>1521</td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1400</td>
</tr>
<tr>
<td>ios_base</td>
<td>1386</td>
</tr>
</tbody>
</table>

Index of library names

<table>
<thead>
<tr>
<th>in</th>
<th>codecvt, 1349</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_avail</td>
<td>basic_streambuf, 1398</td>
</tr>
<tr>
<td>in_place</td>
<td>582, 634</td>
</tr>
<tr>
<td>in_place_index</td>
<td>582</td>
</tr>
<tr>
<td>in_place_index_t</td>
<td>582</td>
</tr>
<tr>
<td>in_place_t</td>
<td>582</td>
</tr>
<tr>
<td>in_place_type</td>
<td>582</td>
</tr>
<tr>
<td>in_range</td>
<td>585</td>
</tr>
<tr>
<td>includes</td>
<td>1127</td>
</tr>
<tr>
<td>inclusive_scan</td>
<td>1149</td>
</tr>
<tr>
<td>increment</td>
<td>directory_iterator, 1487</td>
</tr>
<tr>
<td></td>
<td>recursive_directory_iterator, 1490</td>
</tr>
<tr>
<td>incrementable</td>
<td>938</td>
</tr>
<tr>
<td>incrememt_traits</td>
<td>930</td>
</tr>
<tr>
<td>independent_bits_engine</td>
<td>1187, 1188</td>
</tr>
<tr>
<td>result_type</td>
<td>1188</td>
</tr>
<tr>
<td>index</td>
<td>variant, 619</td>
</tr>
<tr>
<td></td>
<td>weekday_indexed, 1291</td>
</tr>
<tr>
<td></td>
<td>year_month_weekday, 1304</td>
</tr>
<tr>
<td>index_sequence</td>
<td>581</td>
</tr>
<tr>
<td>index_sequence_for</td>
<td>581</td>
</tr>
<tr>
<td>indirect_array</td>
<td>1227</td>
</tr>
<tr>
<td>operator*=</td>
<td>1228</td>
</tr>
<tr>
<td>operator**=</td>
<td>1228</td>
</tr>
<tr>
<td>operator-=</td>
<td>1228</td>
</tr>
<tr>
<td>operator/=</td>
<td>1228</td>
</tr>
<tr>
<td>operator&lt;==</td>
<td>1228</td>
</tr>
<tr>
<td>operator</td>
<td>=</td>
</tr>
<tr>
<td>operator&gt;=</td>
<td>1228</td>
</tr>
<tr>
<td>operator[]</td>
<td>1227</td>
</tr>
<tr>
<td>operatorY=</td>
<td>1228</td>
</tr>
<tr>
<td>operator&amp;=</td>
<td>1228</td>
</tr>
<tr>
<td>operator~=</td>
<td>1228</td>
</tr>
<tr>
<td>operator</td>
<td>=</td>
</tr>
<tr>
<td>value_type</td>
<td>1227</td>
</tr>
<tr>
<td>indirect_binary_predicate</td>
<td>946</td>
</tr>
<tr>
<td>indirect_equivalence_relation</td>
<td>946</td>
</tr>
<tr>
<td>indirect_strict_weak_order</td>
<td>947</td>
</tr>
<tr>
<td>indirect_unary_predicate</td>
<td>946</td>
</tr>
<tr>
<td>indirectly_comparable</td>
<td>948</td>
</tr>
<tr>
<td>indirectly_copyable</td>
<td>948</td>
</tr>
<tr>
<td>indirectly_copyable_storable</td>
<td>948</td>
</tr>
<tr>
<td>indirectly_movable</td>
<td>947</td>
</tr>
<tr>
<td>indirectly_movable_storable</td>
<td>947</td>
</tr>
<tr>
<td>indirectly_readeable</td>
<td>936</td>
</tr>
<tr>
<td>indirectly_readable_traits</td>
<td>931</td>
</tr>
<tr>
<td>indirectly_regular_unary_invocable</td>
<td>946</td>
</tr>
<tr>
<td>indirectly_swappable</td>
<td>948</td>
</tr>
<tr>
<td>indirectly_unary_invocable</td>
<td>946</td>
</tr>
<tr>
<td>indirectly_writable</td>
<td>936</td>
</tr>
<tr>
<td>INFINITY</td>
<td>1229</td>
</tr>
<tr>
<td>infinity</td>
<td>numeric_limits, 516</td>
</tr>
<tr>
<td>Init</td>
<td></td>
</tr>
</tbody>
</table>
Index of library names

ios_base, 1385

init
  basic_ios, 1390, 1406
  basic_ostream, 1417
initializer_list, 537
  begin, 538
  constructor, 538
  end, 538
  size, 538
inner_allocator
  scoped_allocator_adaptor, 684
inner_allocator_type
  scoped_allocator_adaptor, 683
inner_product, 1146
inplace_merge, 1126
input_iterator, 939
input_iterator_tag, 949
input_or_output_iterator, 938
input_range, 992
insert
  basic_string, 780, 781
  deque, 848
  list, 858
  map, 874
  multimap, 879
  ordered associative containers, 822
  unordered associative containers, 834
  unordered_map, 892, 893
  unordered_multimap, 898
  vector, 865
insert_after
  forward_list, 852, 853
insert_iterator, 959
  constructor, 959
  operator*, 959
  operator++, 959
  operator=, 959
insert_or_assign
  map, 875
  unordered_map, 893
inserter, 959
int16_t, 520
int32_t, 520
int64_t, 520
int8_t, 520
int_fast16_t, 520
int_fast32_t, 520
int_fast64_t, 520
int_fast8_t, 520
int_least16_t, 520
int_least32_t, 520
int_least64_t, 520
int_least8_t, 520
INT_MAX, 519
INT_MIN, 519
int_type
  char_traits, 762
  wstring_convert, 1699
integer_sequence, 585

value_type, 585
integral, 558
integral_constant, 715
  value_type, 715
internal, 1394
intervals
  piecewise_constant_distribution, 1208
  piecewise_linear_distribution, 1209
intmax_t, 520
intptr_t, 520
invalid_argument, 566, 567, 630
  constructor, 567
invocable, 564
INVOKES, 687, 688
invoke, 688
invoke_result_t, 712
io_errc, 1379
  make_error_code, 1395
  make_error_condition, 1395
io_state
  zombie, 499
ios, 1376, 1380
ios_base, 1381
  constructor, 1387
  destructor, 1387
  failure, 1383
  flags, 1342, 1385
  fmtflags, 1383, 1423
  getloc, 1386
  imbu, 1386
  Init, 1385
  iostate, 1383
  iword, 1386
  openmode, 1383
  precision, 1342, 1385
  pword, 1387
  register_callback, 1387
  seekdir, 1383
  setf, 1385
  sync_with_stdio, 1386
  unsetf, 1385
  width, 1342, 1385, 1386
  xalloc, 1386
ios_base::failure, 1383
  constructor, 1383
ios_base::Init, 1385
  constructor, 1385
  destructor, 1385
iostate
  ios_base, 1383
  iostream_category, 1395
iota, 1153
iota_view, 1000
  begin, 1002
  constructor, 1002
  end, 1002
  size, 1002
iota_view::iterator, 1002
  constructor, 1003
operator*, 1003
operator+, 1005
operator++, 1004
operator+=, 1004
operator-, 1005
operator-=, 1004
operator--, 1004
operator<, 1005
operator<=, 1005
operator<>, 1005
operator==, 1004
operator>, 1005
operator>=, 1005
operator[], 1004
iota_view::sentinel, 1005
constructor, 1006
ctype, 1344
ctype<char>, 1347
is_absolute
path, 1475
is_abstract
is_abstract_v, 717
is_aggregate
is_aggregate_v, 718
is_always_equal
is_always_lock_free
atomic, 1554
atomic<floating-point>, 1554
atomic<integral>, 1554
atomic<shared_ptr<T>>, 1554
atomic<T>, 1554
atomic<weak_ptr<T>>, 1554
atomic_ref, 1547
atomic_ref<floating-point>, 1547
atomic_ref<integral>, 1547
atomic_ref<T>, 1547
is_am, 1313
is_arithmetic, 716
is_arithmetic_v, 713
is_array, 716
is_array_v, 712
is_assignable, 719
is_assignable_v, 713
is_base_of, 724
is_base_of_v, 715
is_bind_expression, 699
is_bind_expression_v, 686
is_block_file, 1496
directory_entry, 1484
is_bounded
numeric_limits, 517
is_bounded_array, 718
is_bounded_array_v, 713
is_character_file, 1496, 1497
directory_entry, 1484
is_class, 716
is_class_v, 712
is_clock, 1260
is_clock_v, 1245
is_compound, 717
is_compound_v, 713
is_const, 717
is_const_v, 713
is_constructible, 718, 723
is_constructible_v, 713
is_convertible, 724, 725
is_convertible_v, 715
is_copy_assignable, 719
is_copy_assignable_v, 713
is_copy_constructible, 718
is_copy_constructible_v, 713
is_corresponding_member, 732
is_default_constructible, 718
is_default_constructible_v, 713
is_destructible, 720
is_destructible_v, 714
is_directory, 1497
directory_entry, 1485
is_empty
class, 717
function, 1497
is_empty_v, 713
is_enum, 716
is_enum_v, 712
is_eq, 538
is_equal
memory_resource, 673
is_error_code_enum, 571
is_error_condition_enum, 571
is_exact
numeric_limits, 515
is_execution_policy, 738
is_execution_policy_v, 737
is_fifo, 1497
directory_entry, 1485
is_final, 718
is_final_v, 713
is_floating_point, 716
is_floating_point_v, 712
is_function, 716
is_function_v, 712
is_fundamental, 716
is_fundamental_v, 713
is_geq, 538
is_gt, 538
is_gteq, 538
is_heap, 1134
is_heap_until, 1134
is_iec559
numeric_limits, 517
is_integer
numeric_limits, 515
is_integral, 716
Index of library names
is_integral_v, 712
is_invocable, 725
is_invocable_r, 725
is_invocable_r_v, 715
is_invocable_v, 715
is_layout_compatible, 724
is_layout_compatible_v, 715
is_leap
  year, 1287
is_literal_type, 1692
  zombie, 499
is_literal_type_v
  zombie, 499
is_lock_free
  atomic, 1554
    atomic<floating-point>, 1554
    atomic<integral>, 1554
    atomic<shared_ptr<T>>, 1554
    atomic<T>, 1554
    atomic<weak_ptr<T>>, 1554
    atomic_ref, 1547
    atomic_ref<floating-point>, 1547
    atomic_ref<integral>, 1547
    atomic_ref<T*>, 1547
is_lt, 538
is_lteq, 538
is_lvalue_reference, 716
  is_lvalue_reference_v, 712
is_member_function_pointer, 716
is_member_function_pointer_v, 712
is_member_object_pointer, 716
is_member_object_pointer_v, 712
is_member_pointer, 717
is_member_pointer_v, 713
is_modulo
  numeric_limits, 517
is_move_assignable, 719
is_move_assignable_v, 713
is_move_constructible, 718
is_move_constructible_v, 713
is_negative
  hh_mm_ss, 1312
is_neq, 538
is_nothrow_assignable, 722
is_nothrow_assignable_v, 714
is_nothrow_constructible, 721
is_nothrow_convertible, 724
is_nothrow_convertible_v, 715
is_nothrow_copy_assignable, 722
is_nothrow_copy_assignable_v, 714
is_nothrow_copy_constructible, 721
is_nothrow_default_constructible, 721
is_nothrow_destructible, 722
is_nothrow_destructible_v, 714
is_nothrow_invocable, 725
is_nothrow_invocable_r, 725
is_nothrow_invocable_v, 715
is_nothrow_move_assignable, 722
is_nothrow_move_assignable_v, 714
is_nothrow_move_constructible, 721
is_nothrow_swappable, 722
is_nothrow_swappable_v, 714
is_nothrow_swappable_with, 722
is_nothrow_swappable_with_v, 714
is_null_pointer, 716
is_null_pointer_v, 712
is_object, 717
is_object_v, 713
is_open
  basic_filebuf, 1444
  basic_fstream, 1453
  basic_ifstream, 1449
  basic_ofstream, 1450
is_other, 1497
  directory_entry, 1485
is_permutation, 1096
is_placeholder, 700
is_placeholder_v, 686
is.pm, 1313
is_pointer, 716
is_pointer_interconvertible_base_of, 724
is_pointer_interconvertible_with_class, 732
is_pointer_v, 712
is_polymorphic, 717
is_polymorphic_v, 713
is_reference, 716
is_reference_v, 713
is_relative
  path, 1475
  directory_entry, 1485
is_ratereference, 716
is_rvalue_reference_v, 712
is_same_v, 715
is_scalar, 717
is_scalar_v, 713
is_signed
  class, 718
  numeric_limits, 517
is_signed_v, 713
is_socket, 1498
  directory_entry, 1485
is_sorted, 1119
is_sorted_until, 1119
is_standard_layout, 717
is_standard_layout_v, 713
is_swappable, 720
is_swappable_v, 713
is_swappable_with, 720
is_swappable_with_v, 713
is_symlink, 1498
  directory_entry, 1485
is_trivial, 717
is_trivial_v, 713
is_trivially_assignable, 721
is_trivially_assignable_v, 714

Index of library names 1801
J
jmp_buf, 551
join
  jthread, 1584
  thread, 1581
join_view, 1025
  base, 1025
  begin, 1025
  constructor, 1026
  end, 1025
join_view::iterator, 1026
  constructor, 1028
  iter_swap, 1029
  operator++, 1029
  operator->, 1029
  operator--, 1029
  operator==, 1029
join_view::sentinel, 1030
  constructor, 1030
  operator==, 1030
joinable
  jthread, 1584
  thread, 1581
jthread, 1582
  constructor, 1583, 1584
  destructor, 1584
  detach, 1584
  get_id, 1584
  get_stop_source, 1585
  get_stop_token, 1585
  hardware_concurrency, 1585
  join, 1584
  joinable, 1584
  operator==, 1584
  request_stop, 1585
  swap, 1584, 1585

K
k
  negative_binomial_distribution, 1197
key_comp
  ordered associative containers, 821
key_compare
  ordered associative containers, 820
key_eq
  unordered associative containers, 833
key_equal
  unordered associative containers, 830
key_type
  ordered associative containers, 820
  unordered associative containers, 830
kill_dependency, 1545
knuth_b, 1190

L
L_tmpnam, 1503
labs, 507
laguerre, 1242
laguerref, 1242
laguerrel, 1242
lambda
  exponential_distribution, 1199
largest_required_pool_block
  pool_options, 678
last
  span, 919, 920
last_spec, 1282
last_write_time, 1498
  directory_entry, 1485
latch
  arrive_and_wait, 1614
  constructor, 1613
  count_down, 1613
  max, 1613
  try_wait, 1613
  wait, 1614
latest
  choose, 1245
laundry, 529
LC_ALL, 1374
LC_COLLATE, 1374
LC_CTYPE, 1374
LC_MONETARY, 1374
LC_NUMERIC, 1374
LC_TIME, 1374
lcm, 1153
lconv, 1374
LDBL_DECIMAL_DIG, 520
LDBL_DIG, 520
LDBL_EPSILON, 520
LDBL_HAS_SUBNORM, 520
LDBL_MANT_DIG, 520
LDBL_MAX, 520
LDBL_MAX_10_EXP, 520
LDBL_MAX_EXP, 520
LDBL_MIN, 520
LDBL_MIN_10_EXP, 520
LDBL_MIN_EXP, 520
ldexp, 1229
ldexpf, 1229
ldexpl, 1229
ldiv, 507
ldiv_t, 507
leap_second, 1324
  date, 1325
  operator<, 1325
  operator<=, 1325
  operator<> , 1325, 1326
  operator==, 1325
  operator>, 1325
  operator>=, 1325, 1326
  value, 1325
leap_second_info, 1274
left, 1394
legendre, 1242
legendref, 1242

Index of library names 1803
Index of library names

legendrel, 1242
length
  basic_string, 776
  basic_string_view, 795
codecvt, 1349
match_results, 1525
regex_traits, 1515
sub_match, 1521
length_error, 566, 567
  constructor, 567
lerp, 1238
less, 693, 695
  operator(), 693
  partial_ordering, 539
  strong_ordering, 541
  weak_ordering, 540
less<>, 693
  operator(), 693
less_equal, 694, 696
  operator(), 694
less_equal<>, 694
  operator(), 694
lexically_normal
  path, 1475
lexically_proximate
  path, 1476
lexicographically_compare, 1139
lexicographical_compare_three_way, 1140
lgamma, 1229
lgammaf, 1229
lgammal, 1229
linear_congruential_engine, 1183
  constructor, 1183, 1184
  result_type, 1183
list, 855
  constructor, 858
erase, 859, 861
erase_if, 861
insert, 858
merge, 860
remove, 860
resize, 858
reverse, 861
sort, 861
splice, 859, 860
unique, 860
little
  endian, 1173
llabs, 507
ldiv, 507
lldiv_t, 507
LONGLONG_MAX, 519
LONGLONG_MIN, 519
llrint, 1229
llrintf, 1229
llrintl, 1229
llround, 1229
llroundf, 1229
llroundl, 1229
load
  atomic, 1554
  atomic<floating-point>, 1554
  atomic<integral>, 1554
  atomic<shared_ptr<T>>, 1564
  atomic<T*>, 1554
  atomic<weak_ptr<T>>, 1566
  atomic_ref, 1547
  atomic_ref<floating-point>, 1547
  atomic_ref<integral>, 1547
  atomic_ref<T*>, 1547
load_factor
  unordered associative containers, 838
local_time-format-t, 1330
local_days, 1245
local_info, 1318
  ambiguous, 1318
  first, 1318
  nonexistent, 1318
  operator<<, 1318
  result, 1318
  second, 1318
  unique, 1318
local_iterator
  unordered associative containers, 830
local_seconds, 1245
local_t, 1245
local_time, 1245, 1278
  from_stream, 1279
  operator<<, 1278
local_time_format, 1330
locale, 1517, 1521, 1536
  basic_format_context, 754
  category, 1338
  classic, 1342
  combine, 1341
  constructor, 1340, 1341
  facet, 1339
  global, 1341
  has_facet, 1342
  id, 1340
  name, 1341
  operator(), 1341
  operator=, 1341
  operator==, 1341
  use_facet, 1342
localeconv, 1374
localtime, 1334
locate_zone, 1315
tzdb, 1314
  zoned_traits<const time_zone*>, 1320
lock, 1602
  shared_lock, 1601
  unique_lock, 1597
  weak_ptr, 670
lock_guard, 1594
  constructor, 1594
Index of library names 1805

M
m

fisher_f_distribution, 1205
lognormal_distribution, 1202
make12, 1313
make24, 1313
make_any, 627
make_error_code
  errc, 576
  future_errc, 1617
  io_errc, 1395
make_error_condition
  errc, 577
  future_errc, 1617
  io_errc, 1395
make_exception_ptr, 536
make_format_args, 758
make_from_tuple, 597
make_heap, 1133
make_index_sequence, 581
make_integer_sequence, 585
make_move_iterator, 963
make_obj_using_allocator, 645
make_optional, 611, 612
make_pair, 589
make_preferred
  path, 1471
make_ready_at_thread_exit
  packaged_task, 1629
make_reverse_iterator, 956
make_shared, 664–666
make_signature, 727
make_signed, 1202
make_tuple, 596
tuple, 596
make_unique, 656, 657
make_unsigned, 727
make_unsigned_t, 711
make_wformat_args, 758
malloc, 507, 649, 1680
map, 870
at, 874
clear, 826
constructor, 821, 873
contains, 826
count, 826
emplace, 822
emplace_hint, 822
equal_range, 827
erase, 826
erase_if, 875
extract, 825
find, 826
insert, 822, 874
insert_or_assign, 875
key_compare, 821
key_compare, 820
key_type, 820

log, 1229
  complex, 1169
  valarray, 1221
log10, 1229
  complex, 1169
  valarray, 1221
log10f, 1229
log10l, 1229
log1p, 1229
log1pf, 1229
log1pl, 1229
log2, 1229
log2f, 1229
log2l, 1229
logb, 1229
logbf, 1229
logbl, 1229
logf, 1229
log_error, 566
  constructor, 566, 567
logical_and, 696
  operator(), 696
logical_and<>, 696
  operator(), 696
logical_not, 697
  operator(), 697
logical_not<>, 697
  operator(), 697
logical_or, 696
  operator(), 696
logical_or<>, 696
  operator(), 697
log1, 1229
lognormal_distribution, 1202
  constructor, 1202
  m, 1202
  result_type, 1202
  s, 1202
LONG_MAX, 519
LONG_MIN, 519
longjmp, 551

lookup_classname
  regex_traits, 1516
  regular_expression_traits, 1537
lookup_colonename
  regex_traits, 1515
  regular_expression_traits, 1537
lower_bound, 1121
  ordered associative containers, 827
lowest
  numeric_limits, 515
lrint, 1229
lrintf, 1229
lrintl, 1229
lround, 1229
lroundf, 1229
lroundl, 1229

Index of library names 1805
index of library names
poisson_distribution, 1198
student_t_distribution, 1205
mem_fn, 701
mem_fun
  zombie, 499
mem_fun1_ref_t
  zombie, 499
mem_fun1_t
  zombie, 499
mem_fun_ref
  zombie, 499
mem_fun_ref_t
  zombie, 499
mem_fun_t
  zombie, 499
memchr, 801
memcmp, 801
memcpy, 801
memmove, 801
memory_order, 1543
  acq_rel, 1543
  acquire, 1543
  consume, 1543
  relaxed, 1543
  release, 1543
  seq_cst, 1543
memory_order_acq_rel, 1543
memory_order_acquire, 1543
memory_order_consume, 1543
memory_order_relaxed, 1543
memory_order_release, 1543
memory_order_seq_cst, 1543
memory_resource, 673
  allocate, 673
  deallocate, 673
  destructor, 673
  do_allocate, 673
  do_deallocate, 673
  is_equal, 674
  is_equal, 673
  operator=, 673
  operator==, 674
memset, 801
merge, 1125
  forward_list, 855
  list, 860
    ordered associative containers, 825
    unordered associative containers, 836
mergeable, 949
mersenne_twister_engine, 1184
  constructor, 1185
  result_type, 1184
message
  do_close, 1373
  error_category, 574
  error_code, 576
  error_condition, 577
messages, 1372
  close, 1373
do_get, 1373
do_open, 1373
get, 1373
open, 1373
messages_byname, 1374
microseconds, 1245
midpoint, 1154
milliseconds, 1245
min, 1135
duration, 1263
duration_values, 1260
numeric_limits, 514
time_point, 1269
valarray, 1219
year, 1287
min_element, 1137
min_exponent
  numeric_limits, 516
min_exponent10
  numeric_limits, 516
minmax, 1136, 1137
minmax_element, 1138
minstd_rand, 1189
minstd_rand0, 1189
minus, 690
  operator(), 690
  minus<>, 690
    operator(), 691
minutes, 1245
  hh_mm_ss, 1312
mismatch, 1093
mktime, 1334
modf, 1229
modff, 1229
modfl, 1229
modulus, 691
    operator(), 691
    modulus<>, 691
    operator(), 691
money_get, 1367
  do_get, 1368
  get, 1368
money_put, 1369
  do_put, 1369
  put, 1369
moneypunct, 1370
    curr_symbol, 1371
    decimal_point, 1371
do_curr_symbol, 1372
do_decimal_point, 1371
do_frac_digits, 1372
do_grouping, 1372
do_neg_format, 1372
do_negative_sign, 1372
do_pos_format, 1372
do_positive_sign, 1372
do_thousands_sep, 1371
frac_digits, 1371
grouping, 1371

Index of library names 1807
negative_sign, 1371
positive_sign, 1371
thousands_sep, 1371
moneypunct_byname, 1372
monostate, 622
hash, 623
operator<<, 622
operator==, 622
monotonic_buffer_resource, 680
constructor, 680
destructor, 680
do_allocate, 681
do_deallocate, 681
do_is_equal, 681
release, 680
upstream_resource, 681
month, 1284
constructor, 1284
from_stream, 1285
month_day, 1292
month_day_last, 1293
month_weekday, 1294
month_weekday_last, 1295
ok, 1285
operator unsigned, 1285
operator+, 1285
operator++, 1284
operator=, 1284
operator-, 1285
operator--, 1284
operator=, 1284
operator<<, 1285
operator>>, 1285
operator==, 1285
year_month, 1296
year_month_day, 1299
year_month_day_last, 1302
year_month_weekday, 1304
year_month_weekday_last, 1306
month_day, 1292
constructor, 1292
day, 1292
from_stream, 1293
month, 1292
ok, 1292
operator<<, 1293
operator>>, 1293
operator==, 1293
month_day_last, 1293
constructor, 1293
month, 1293
ok, 1293
operator<<, 1294
operator>>, 1294
operator==, 1294
year_month_day_last, 1302
month_weekday, 1294
constructor, 1294
month, 1294
ok, 1294
operator<<, 1294
operator==, 1294
weekday_indexed, 1294
month_weekday_last, 1295
constructor, 1295
month, 1295
ok, 1295
operator<<, 1295
operator==, 1295
weekday_last, 1295
months, 1245
movable, 564
move
algorithm, 1101
basic_ios, 1391
function, 583
move_backward, 1101
move_constructible, 561
move_if_noexcept, 584
move_iterator, 960
base, 961
constructor, 961
iter_move, 963
iter_swap, 963
operator*, 961
operator+, 962, 963
operator++, 962
operator+=, 962
operator=, 962
operator->, 964
operator==, 962
operator<, 962
operator<=, 962
operator<=, 963
operator<, 963
operator==, 963
operator=, 961
operator=, 962
operator->, 962
operator=, 962
operator=, 963
operator=, 963
operator=, 963
operator=, 963
operator=, 961
operator=, 962
operator=, 962
operator=, 963
operator=, 963
move_sentinel, 964
base, 964
constructor, 964
operator=, 964
mt19937, 1189
mt19937_64, 1189
multiline
syntax_option_type, 1512
multimap, 587
clear, 826
constructor, 821, 878
contains, 826
count, 826
emplace, 822
emplace_hint, 822
equal_range, 827
erase, 826
erase_if, 879

Index of library names 1808
Index of library names

character

chi_squared_distribution, 1203
fisher_f_distribution, 1205
name
nexttowardl, 1229
noboolalpha, 1393
node_type
  ordered associative containers, 820
  unordered associative containers, 830
noemit_on_flush, 1422
none
  bitset, 632
  none_of, 1087
nonexistent
  local_info, 1318
  nonexistent_local_time, 1316
    constructor, 1316
noop_coroutine, 550
noop_coroutine_handle, 546
noop_coroutine_promise, 549
norm, 1170
  complex, 1168
  normal_distribution, 1201
    constructor, 1202
    result_type, 1201
stddev, 1202
noshowbase, 1393
noshovpoint, 1393
noshovpos, 1393
noskipws, 1393
nostopstate, 1575
nostopstate_t, 1575
nosubs
  syntax_option_type, 1512
not1
  zombie, 499
not2
  zombie, 499
not_equal_to, 692, 695
  operator(), 692
not_equal_to<>, 693
  operator(), 693
not_fn, 699
nothrow, 523
nothrow_t, 523
notify_all
  atomic, 1557
  atomic<floating-point>, 1557
  atomic<integral>, 1557
  atomic<shared_ptr<T>*, 1565
  atomic<T>, 1557
  atomic<weak_ptr<T>>, 1567
  atomic_ref<T>, 1548
  condition_variable, 1605
  condition_variable_any, 1608
nounitbuf, 1393
nouppercase, 1393
now
  file_clock, 1277
  gps_clock, 1276
  tai_clock, 1275
  utc_clock, 1273
nth_element, 1120
NULL, 506–508, 802, 1334, 1374, 1503
null_memory_resource, 676
nullopt, 609
nullptr_t, 609
nullptr_t<T>, 506, 508
num_get, 1352
  do_get, 1353, 1355
  get, 1353
num_put, 1355
  do_put, 1356, 1358
  put, 1356
numeric_limits, 512, 513
  denorm_min, 517
  digits, 515
  digits10, 515
  epsilon, 515
  float_denorm_style, 516
  has_denorm_loss, 516
  has_infinity, 516
  has_quiet_NaN, 516
  has_signaling_NaN, 516
  infinity, 516
  is_bounded, 517
  is_exact, 515
  is_iec559, 517
  is_integer, 515
  is_modulo, 517
  is_signed, 515
  lowest, 515
  max, 515
  max_digits10, 515
  max_exponent, 516
  max_exponent10, 516
  min, 514
  min_exponent, 516
  min_exponent10, 516
  quiet_NaN, 517
  radix, 515
  round_error, 515
  round_style, 517
  signaling_NaN, 517
  tinyness_before, 517
  traps, 517
numeric_limits<false>, 518
numpunct, 1359
  decimal_point, 1360
  do_decimal_point, 1360

Index of library names 1810
Index of library names

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC

© ISO/IEC
Index of library names

© ISO/IEC N4868

operator!
  basic_ios, 1392
  valarray, 1217
operator!=, 1684
  coroutine_handle, 549
  optional, 610, 611
  queue, 910
  reverse_iterator, 955
  stack, 915
  valarray, 1220, 1221
  variant, 621
operator**d
  day, 1283
operator**h
  duration, 1266
operator**i
  complex, 1170
operator**if
  complex, 1170
operator**il
  complex, 1170
operator**min
  duration, 1266
operator**ms
  duration, 1266
operator**ns
  duration, 1266
operator**s
  duration, 1266
  string, 791
  u16string, 791
  u32string, 791
  u8string, 791
  wstring, 791
operator**sv
  string_view, 800
  u16string_view, 800
  u32string_view, 800
  u8string_view, 800
  wstring_view, 800
operator**us
  duration, 1266
operator**y
  year, 1288
operator()
  bit_and, 697
  bit_and<>, 697
  bit_not, 698
  bit_not<>, 698
  bit_or, 697
  bit_or<>, 698
  bit_xor, 698
  bit_xor<>, 698
  boyer_moore_horspool_searcher, 707
  boyer_moore_searcher, 706
  clock_time_conversion, 1279–1281
  coroutine_handle, 548
  coroutine_handle<noop_coroutine_<
    promise>, 549
  default_delete, 650
  default_searcher, 705
divides, 691
divides<>, 691
equal_to, 692
equal_to<>, 692
function, 704
greater, 693
greater<>, 693
greater_equal, 693
greater_equal<>, 694
less, 693
less<>, 693
less_equal, 694
less_equal<>, 694
locale, 1341
logical_and, 696
logical_and<>, 696
logical_not, 697
logical_not<>, 697
logical_or, 696
logical_or<>, 697
map::value_compare, 870
minus, 690
minus<>, 691
modulus, 691
modulus<>, 691
multimap::value_compare, 875
multiplies, 691
multiplies<>, 691
negate, 692
negate<>, 692
not_equal_to, 692
not_equal_to<>, 693
owner_less, 671
packaged_task, 1629
plus, 690
plus<>, 690
random_device, 1191
reference_wrapper, 690
reference_wrapper, 690
stringbuf_iterator, 977

Index of library names

1812
<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ostreambuf_iterator</td>
<td>978</td>
</tr>
<tr>
<td>regex_iterator</td>
<td>1533</td>
</tr>
<tr>
<td>regex_token_iterator</td>
<td>1536</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>954</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>663</td>
</tr>
<tr>
<td>split_view::outer_iterator</td>
<td>1033</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>653</td>
</tr>
<tr>
<td>valarray</td>
<td>1219, 1220</td>
</tr>
<tr>
<td>operator**</td>
<td></td>
</tr>
<tr>
<td>complex, 1166, 1167</td>
<td></td>
</tr>
<tr>
<td>duration, 1263</td>
<td></td>
</tr>
<tr>
<td>gslice_array, 1226</td>
<td></td>
</tr>
<tr>
<td>indirect_array, 1228</td>
<td></td>
</tr>
<tr>
<td>mask_array, 1227</td>
<td></td>
</tr>
<tr>
<td>slice_array, 1223</td>
<td></td>
</tr>
<tr>
<td>valarray, 1218</td>
<td></td>
</tr>
<tr>
<td>operator+</td>
<td></td>
</tr>
<tr>
<td>basic_string, 786, 787</td>
<td></td>
</tr>
<tr>
<td>complex, 1167</td>
<td></td>
</tr>
<tr>
<td>counted_iterator, 971</td>
<td></td>
</tr>
<tr>
<td>day, 1283</td>
<td></td>
</tr>
<tr>
<td>duration, 1262, 1269</td>
<td></td>
</tr>
<tr>
<td>elements_view::iterator, 1042</td>
<td></td>
</tr>
<tr>
<td>iota_view::iterator, 1005</td>
<td></td>
</tr>
<tr>
<td>month, 1285</td>
<td></td>
</tr>
<tr>
<td>move_iterator, 962, 963</td>
<td></td>
</tr>
<tr>
<td>reverse_iterator, 954, 956</td>
<td></td>
</tr>
<tr>
<td>time_point, 1269</td>
<td></td>
</tr>
<tr>
<td>transform_view::iterator, 1018</td>
<td></td>
</tr>
<tr>
<td>valarray, 1217, 1219, 1220</td>
<td></td>
</tr>
<tr>
<td>weekday, 1289, 1290</td>
<td></td>
</tr>
<tr>
<td>year, 1287</td>
<td></td>
</tr>
<tr>
<td>year_month, 1297</td>
<td></td>
</tr>
<tr>
<td>year_month_day, 1300</td>
<td></td>
</tr>
<tr>
<td>year_month_day_last, 1302, 1303</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday, 1305</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday_last, 1307</td>
<td></td>
</tr>
<tr>
<td>operator++</td>
<td></td>
</tr>
<tr>
<td>atomic&lt;integral&gt;, 1562</td>
<td></td>
</tr>
<tr>
<td>atomic&lt;T*&gt;, 1562</td>
<td></td>
</tr>
<tr>
<td>atomic_ref&lt;integral&gt;, 1552</td>
<td></td>
</tr>
<tr>
<td>atomic_ref&lt;T*&gt;, 1552</td>
<td></td>
</tr>
<tr>
<td>back_insert_iterator, 958</td>
<td></td>
</tr>
<tr>
<td>basic_istream_view::iterator, 1007</td>
<td></td>
</tr>
<tr>
<td>common_iterator, 967</td>
<td></td>
</tr>
<tr>
<td>counted_iterator, 971</td>
<td></td>
</tr>
<tr>
<td>day, 1282</td>
<td></td>
</tr>
<tr>
<td>directory_iterator, 1487</td>
<td></td>
</tr>
<tr>
<td>duration, 1263</td>
<td></td>
</tr>
<tr>
<td>elements_view::iterator, 1041</td>
<td></td>
</tr>
<tr>
<td>filter_view::iterator, 1012</td>
<td></td>
</tr>
<tr>
<td>front_insert_iterator, 958</td>
<td></td>
</tr>
<tr>
<td>insert_iterator, 959</td>
<td></td>
</tr>
<tr>
<td>iota_view::iterator, 1004</td>
<td></td>
</tr>
<tr>
<td>istream_iterator, 974</td>
<td></td>
</tr>
<tr>
<td>istreambuf_iterator, 977</td>
<td></td>
</tr>
<tr>
<td>join_view::iterator, 1029</td>
<td></td>
</tr>
<tr>
<td>month, 1284</td>
<td></td>
</tr>
<tr>
<td>move_iterator, 962</td>
<td></td>
</tr>
<tr>
<td>ostream_iterator, 976</td>
<td></td>
</tr>
<tr>
<td>ostreambuf_iterator, 978</td>
<td></td>
</tr>
<tr>
<td>recursive_directory_iterator, 1490</td>
<td></td>
</tr>
<tr>
<td>regex_iterator, 1533</td>
<td></td>
</tr>
<tr>
<td>regex_token_iterator, 1536</td>
<td></td>
</tr>
<tr>
<td>reverse_iterator, 954, 955</td>
<td></td>
</tr>
<tr>
<td>split_view::inner_iterator, 1035</td>
<td></td>
</tr>
<tr>
<td>split_view::outer_iterator, 1033</td>
<td></td>
</tr>
<tr>
<td>time_point, 1269</td>
<td></td>
</tr>
<tr>
<td>transform_view::iterator, 1017</td>
<td></td>
</tr>
<tr>
<td>weekday, 1289</td>
<td></td>
</tr>
<tr>
<td>year, 1286</td>
<td></td>
</tr>
<tr>
<td>operator+=</td>
<td></td>
</tr>
<tr>
<td>atomic&lt;floating-point&gt;, 1560</td>
<td></td>
</tr>
<tr>
<td>atomic&lt;integral&gt;, 1559</td>
<td></td>
</tr>
<tr>
<td>atomic&lt;T*&gt;, 1559, 1560, 1562</td>
<td></td>
</tr>
<tr>
<td>atomic_ref&lt;floating-point&gt;, 1551</td>
<td></td>
</tr>
<tr>
<td>atomic_ref&lt;integral&gt;, 1550</td>
<td></td>
</tr>
<tr>
<td>atomic_ref&lt;T*&gt;, 1552</td>
<td></td>
</tr>
<tr>
<td>basic_string, 777, 778</td>
<td></td>
</tr>
<tr>
<td>complex, 1166</td>
<td></td>
</tr>
<tr>
<td>counted_iterator, 971</td>
<td></td>
</tr>
<tr>
<td>day, 1283</td>
<td></td>
</tr>
<tr>
<td>duration, 1263</td>
<td></td>
</tr>
<tr>
<td>elements_view::iterator, 1042</td>
<td></td>
</tr>
<tr>
<td>gslice_array, 1226</td>
<td></td>
</tr>
<tr>
<td>indirect_array, 1228</td>
<td></td>
</tr>
<tr>
<td>iota_view::iterator, 1004</td>
<td></td>
</tr>
<tr>
<td>mask_array, 1227</td>
<td></td>
</tr>
<tr>
<td>month, 1284</td>
<td></td>
</tr>
<tr>
<td>move_iterator, 962</td>
<td></td>
</tr>
<tr>
<td>path, 1471</td>
<td></td>
</tr>
<tr>
<td>reverse_iterator, 955</td>
<td></td>
</tr>
<tr>
<td>slice_array, 1223</td>
<td></td>
</tr>
<tr>
<td>time_point, 1269</td>
<td></td>
</tr>
<tr>
<td>transform_view::iterator, 1017</td>
<td></td>
</tr>
<tr>
<td>valarray, 1218</td>
<td></td>
</tr>
<tr>
<td>weekday, 1289</td>
<td></td>
</tr>
<tr>
<td>year, 1287</td>
<td></td>
</tr>
<tr>
<td>year_month, 1296</td>
<td></td>
</tr>
<tr>
<td>year_month_day, 1299</td>
<td></td>
</tr>
<tr>
<td>year_month_day_last, 1301</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday, 1304</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday_last, 1306</td>
<td></td>
</tr>
<tr>
<td>operator-</td>
<td></td>
</tr>
<tr>
<td>common_iterator, 968</td>
<td></td>
</tr>
<tr>
<td>complex, 1167</td>
<td></td>
</tr>
<tr>
<td>counted_iterator, 972</td>
<td></td>
</tr>
<tr>
<td>day, 1283</td>
<td></td>
</tr>
<tr>
<td>duration, 1262, 1269</td>
<td></td>
</tr>
<tr>
<td>elements_view::iterator, 1042, 1043</td>
<td></td>
</tr>
<tr>
<td>elements_view::sentinel, 1043</td>
<td></td>
</tr>
<tr>
<td>iota_view::iterator, 1005</td>
<td></td>
</tr>
<tr>
<td>month, 1285</td>
<td></td>
</tr>
<tr>
<td>move_iterator, 962, 963</td>
<td></td>
</tr>
<tr>
<td>reverse_iterator, 954, 956</td>
<td></td>
</tr>
<tr>
<td>time_point, 1269</td>
<td></td>
</tr>
<tr>
<td>transform_view::iterator, 1018</td>
<td></td>
</tr>
<tr>
<td>transform_view::sentinel, 1019</td>
<td></td>
</tr>
</tbody>
</table>
valarray, 1217, 1219, 1220
weekday, 1290
year, 1287
year_month, 1297
year_month_day, 1300
year_month_day_last, 1302, 1303
year_month_weekday, 1305
year_month_weekday_last, 1307
operator--
  day, 1282
  elements_view::iterator, 1042
  month, 1284
  time_point, 1269
weekday, 1289
year, 1286
operator=
  atomic<floating-point>, 1560
  atomic<integral>, 1559
  atomic<T*>, 1559, 1560, 1562
  atomic_ref<floating-point>, 1551
  atomic_ref<integral>, 1550
  atomic_ref<T*>, 1552
  complex, 1166, 1167
  counted_iterator, 972
day, 1283
duration, 1263
  elements_view::iterator, 1042
gslice_array, 1226
  indirect_array, 1228
  iota_view::iterator, 1004
mask_array, 1227
month, 1284
move_iterator, 962
reversed_iterator, 955
slice_array, 1223
time_point, 1269
transform_view::iterator, 1017
valarray, 1218
weekday, 1289
year, 1287
year_month, 1296
year_month_day, 1299
year_month_day_last, 1301, 1302
year_month_weekday, 1304
year_month_weekday_last, 1306
operator->
  common_iterator, 967
  filter_view::iterator, 1012
  istream_iterator, 974
  join_view::iterator, 1029
  move_iterator, 1694
  optional, 608
  regex_iterator, 1533
  regex_token_iterator, 1536
  reversed_iterator, 954
  shared_ptr, 663
  unique_ptr, 654
operator--
  atomic<integral>, 1562
  atomic<T*>, 1562
  atomic_ref<integral>, 1552
  atomic_ref<T*>, 1552
duration, 1263
  filter_view::iterator, 1012
  iota_view::iterator, 1004
  join_view::iterator, 1029
move_iterator, 962
reversed_iterator, 955
transform_view::iterator, 1017
operator/
calendar types, 1308–1311
complex, 1167
duration, 1264
path, 1478
valarray, 1219, 1220
operator/=complex, 1166, 1167
duration, 1263
gslice_array, 1226
indirect_array, 1228
mask_array, 1227
path, 1470, 1471
slice_array, 1223
valarray, 1218
operator<
duration, 1265
elements_view::iterator, 1042
iota_view::iterator, 1005
leap_second, 1325
map, 873
move_iterator, 962
multimap, 878
multiset, 885
optional, 610, 611
partial_ordering, 540
queue, 910
reversed_iterator, 955
set, 882
stack, 915
strong_ordering, 542
sys_time, 1325
time_point, 1270
transform_view::iterator, 1017
type_index, 736
unique_ptr, 657, 658
valarray, 1220, 1221
variant, 621
vector, 863
weak_ordering, 541
operator<<
basic_ostream, 1419–1422
basic_string, 788
basic_string_view, 800
bitset, 632, 633
byte, 509
complex, 1168
day, 1283
duration, 1267
error_code, 576
file_time, 1277
gps_time, 1276
local_info, 1318
local_time, 1278
month, 1285
month_day, 1293
month_day_last, 1294
month_weekday, 1294
month_weekday_last, 1295
path, 1477
shared_ptr, 668
sub_match, 1523
sys_days, 1272
sys_info, 1318
sys_time, 1271
tai_time, 1276
thread::id, 1580
unique_ptr, 658
utc_time, 1273
valarray, 1219, 1220
weekday, 1290
weekday_indexed, 1291
weekday_last, 1292
year, 1287
year_month, 1297
year_month_day, 1300
year_month_day_last, 1303
year_month_weekday, 1305
year_month_weekday_last, 1307
zoned_time, 1324
operator>>=
  bitset, 631
byte, 509
gslice_array, 1226
indirect_array, 1228
mask_array, 1227
slice_array, 1223
valarray, 1218
operator<<
  duration, 1265
elements_view::iterator, 1042
iota_view::iterator, 1005
leap_second, 1325
move_iterator, 963
optional, 610, 611
partial_ordering, 540
queue, 910
reverse_iterator, 955
stack, 915
strong_ordering, 542
sys_time, 1325
time_point, 1270
transform_view::iterator, 1017
type_index, 736
unique_ptr, 657, 658
valarray, 1220, 1221
variant, 621
weak_ordering, 541
operator<<=
  basic_string_view, 800
coroutine_handle, 549
counted_iterator, 972
day, 1283
directory_entry, 1486
duration, 1265
elements_view::iterator, 1042
error_category, 574
error_code, 578
error_condition, 578
iota_view::iterator, 1005
leap_second, 1325, 1326
monostate, 622
month, 1285
month_day, 1293
month_day_last, 1294
move_iterator, 963
optional, 610, 611
pair, 589
partial_ordering, 540
path, 1477
queue, 911
reverse_iterator, 956
shared_ptr, 667
stack, 915
strong_ordering, 542
sub_match, 1522, 1523
sys_time, 1326
thread::id, 1580
time_zone, 1320
time_zone_link, 1326
transform_view::iterator, 1018
tuple, 599
type_index, 736
unique_ptr, 657, 658
variant, 621
weak_ordering, 541
year, 1287
year_month, 1297
year_month_day, 1300
year_month_day_last, 1302
year_month_day, 1300
year_month_day_last, 1302
operator=
  allocator, 648
any, 625, 626
atomic, 1554
atomic<floating-point>, 1554
atomic<integral>, 1554
atomic<shared_ptr<T>>, 1564
atomic<T*>, 1554
atomic<weak_ptr<T>>, 1566
atomic_ref, 1547
atomic_ref<floating-point>, 1547
atomic_ref<integral>, 1547
atomic_ref<T*>, 1547
back_insert_iterator, 957
basic_filebuf, 1443
basic_istream, 1414
basic_ostream, 1406

Index of library names 1815
<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic_ostream</td>
<td>1417</td>
</tr>
<tr>
<td>basic_regex</td>
<td>1520</td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1399</td>
</tr>
<tr>
<td>basic_string</td>
<td>775</td>
</tr>
<tr>
<td>basic_stringbuf</td>
<td>1430</td>
</tr>
<tr>
<td>common_iterator</td>
<td>966</td>
</tr>
<tr>
<td>coroutine_handle</td>
<td>548</td>
</tr>
<tr>
<td>counted_iterator</td>
<td>970</td>
</tr>
<tr>
<td>directory_iterator</td>
<td>1487</td>
</tr>
<tr>
<td>enable_shared_from_this</td>
<td>672</td>
</tr>
<tr>
<td>error_code</td>
<td>575</td>
</tr>
<tr>
<td>error_condition</td>
<td>577</td>
</tr>
<tr>
<td>exception</td>
<td>534</td>
</tr>
<tr>
<td>front_insert_iterator</td>
<td>958</td>
</tr>
<tr>
<td>function</td>
<td>703, 704</td>
</tr>
<tr>
<td>future</td>
<td>1622</td>
</tr>
<tr>
<td>gslice_array</td>
<td>1226</td>
</tr>
<tr>
<td>indirect_array</td>
<td>1228</td>
</tr>
<tr>
<td>insert_iterator</td>
<td>959</td>
</tr>
<tr>
<td>jthread</td>
<td>1584</td>
</tr>
<tr>
<td>locale</td>
<td>1341</td>
</tr>
<tr>
<td>mask_array</td>
<td>1227</td>
</tr>
<tr>
<td>match_results</td>
<td>1524, 1525</td>
</tr>
<tr>
<td>memory_resource</td>
<td>673</td>
</tr>
<tr>
<td>move_iterator</td>
<td>961</td>
</tr>
<tr>
<td>move_sentinel</td>
<td>964</td>
</tr>
<tr>
<td>optional</td>
<td>604–606</td>
</tr>
<tr>
<td>ostream_iterator</td>
<td>975</td>
</tr>
<tr>
<td>ostreambuf_iterator</td>
<td>978</td>
</tr>
<tr>
<td>packaged_task</td>
<td>1628</td>
</tr>
<tr>
<td>pair</td>
<td>588</td>
</tr>
<tr>
<td>path</td>
<td>1470</td>
</tr>
<tr>
<td>promise</td>
<td>1620</td>
</tr>
<tr>
<td>recursive_directory_iterator</td>
<td>1489</td>
</tr>
<tr>
<td>reference_wrapper</td>
<td>680</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>954</td>
</tr>
<tr>
<td>shared_future</td>
<td>1624</td>
</tr>
<tr>
<td>shared_lock</td>
<td>1600</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>662</td>
</tr>
<tr>
<td>slice_array</td>
<td>1223</td>
</tr>
<tr>
<td>span</td>
<td>919</td>
</tr>
<tr>
<td>stop_source</td>
<td>1576</td>
</tr>
<tr>
<td>stop_token</td>
<td>1575</td>
</tr>
<tr>
<td>thread</td>
<td>1581</td>
</tr>
<tr>
<td>tuple</td>
<td>595, 596</td>
</tr>
<tr>
<td>unique_lock</td>
<td>1597</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>653, 656</td>
</tr>
<tr>
<td>valarray</td>
<td>1215</td>
</tr>
<tr>
<td>variant</td>
<td>617</td>
</tr>
<tr>
<td>weak_ptr</td>
<td>670</td>
</tr>
<tr>
<td>zoned_time</td>
<td>1323</td>
</tr>
<tr>
<td>operator==</td>
<td></td>
</tr>
<tr>
<td>allocator</td>
<td>649</td>
</tr>
<tr>
<td>basic_istream::iterator</td>
<td>1008</td>
</tr>
<tr>
<td>basic_string_view</td>
<td>800</td>
</tr>
<tr>
<td>bitset</td>
<td>632</td>
</tr>
<tr>
<td>common_iterator</td>
<td>967</td>
</tr>
<tr>
<td>complex</td>
<td>1167</td>
</tr>
<tr>
<td>coroutine_handle</td>
<td>549</td>
</tr>
<tr>
<td>counted_iterator</td>
<td>972</td>
</tr>
<tr>
<td>day</td>
<td>1283</td>
</tr>
<tr>
<td>directory_entry</td>
<td>1486</td>
</tr>
<tr>
<td>duration</td>
<td>1264</td>
</tr>
<tr>
<td>elements_view::iterator</td>
<td>1042</td>
</tr>
<tr>
<td>elements_view::sentinel</td>
<td>1043</td>
</tr>
<tr>
<td>error_category</td>
<td>574</td>
</tr>
<tr>
<td>error_code</td>
<td>577, 578</td>
</tr>
<tr>
<td>error_condition</td>
<td>578</td>
</tr>
<tr>
<td>filter_view::iterator</td>
<td>1012</td>
</tr>
<tr>
<td>filter_view::sentinel</td>
<td>1013</td>
</tr>
<tr>
<td>function</td>
<td>704</td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>1004</td>
</tr>
<tr>
<td>iota_view::sentinel</td>
<td>1006</td>
</tr>
<tr>
<td>istream_iterator</td>
<td>974, 975</td>
</tr>
<tr>
<td>istreambuf_iterator</td>
<td>977</td>
</tr>
<tr>
<td>join_view::iterator</td>
<td>1029</td>
</tr>
<tr>
<td>join_view::sentinel</td>
<td>1030</td>
</tr>
<tr>
<td>leap_second</td>
<td>1325</td>
</tr>
<tr>
<td>locale</td>
<td>1341</td>
</tr>
<tr>
<td>map</td>
<td>873</td>
</tr>
<tr>
<td>match_results</td>
<td>1527</td>
</tr>
<tr>
<td>memory_resource</td>
<td>674</td>
</tr>
<tr>
<td>monostate</td>
<td>622</td>
</tr>
<tr>
<td>month</td>
<td>1285</td>
</tr>
<tr>
<td>month_day</td>
<td>1293</td>
</tr>
<tr>
<td>month_day_last</td>
<td>1294</td>
</tr>
<tr>
<td>month_weekday</td>
<td>1294</td>
</tr>
<tr>
<td>month_weekday_last</td>
<td>1295</td>
</tr>
<tr>
<td>move_iterator</td>
<td>962</td>
</tr>
<tr>
<td>multimap</td>
<td>878</td>
</tr>
<tr>
<td>multiset</td>
<td>885</td>
</tr>
<tr>
<td>optional</td>
<td>609, 610</td>
</tr>
<tr>
<td>pair</td>
<td>588</td>
</tr>
<tr>
<td>partial_ordering</td>
<td>540</td>
</tr>
<tr>
<td>path</td>
<td>1477</td>
</tr>
<tr>
<td>polymorphic_allocator</td>
<td>676</td>
</tr>
<tr>
<td>queue</td>
<td>910</td>
</tr>
<tr>
<td>regex_iterator</td>
<td>1532</td>
</tr>
<tr>
<td>regex_token_iterator</td>
<td>1534, 1536</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>955</td>
</tr>
<tr>
<td>scoped_allocator_adaptor</td>
<td>684</td>
</tr>
<tr>
<td>set</td>
<td>882</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>667</td>
</tr>
<tr>
<td>split_view::inner-iterator</td>
<td>1035</td>
</tr>
<tr>
<td>split_view::outer-iterator</td>
<td>1033</td>
</tr>
<tr>
<td>stack</td>
<td>915</td>
</tr>
<tr>
<td>stop_source</td>
<td>1577</td>
</tr>
<tr>
<td>stop_token</td>
<td>1575</td>
</tr>
<tr>
<td>strong_ordering</td>
<td>542</td>
</tr>
<tr>
<td>sub_match</td>
<td>1522</td>
</tr>
<tr>
<td>sys_time</td>
<td>1325</td>
</tr>
<tr>
<td>take_view::sentinel</td>
<td>1021</td>
</tr>
<tr>
<td>take_while_view::sentinel</td>
<td>1023</td>
</tr>
<tr>
<td>thread::id</td>
<td>1580</td>
</tr>
<tr>
<td>time_point</td>
<td>1270</td>
</tr>
<tr>
<td>time_zone</td>
<td>1320</td>
</tr>
<tr>
<td>time_zone_link</td>
<td>1326</td>
</tr>
<tr>
<td>Library Name</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>1017</td>
</tr>
<tr>
<td>transform_view::sentinel</td>
<td>1019</td>
</tr>
<tr>
<td>tuple</td>
<td>599</td>
</tr>
<tr>
<td>type_index</td>
<td>736</td>
</tr>
<tr>
<td>type_info</td>
<td>530</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>657</td>
</tr>
<tr>
<td>unreachable_sentinel_t</td>
<td>973</td>
</tr>
<tr>
<td>valarray</td>
<td>1220, 1221</td>
</tr>
<tr>
<td>variant</td>
<td>621</td>
</tr>
<tr>
<td>vector</td>
<td>863</td>
</tr>
<tr>
<td>weak_ordering</td>
<td>541</td>
</tr>
<tr>
<td>weekday</td>
<td>1289</td>
</tr>
<tr>
<td>weekday_indexed</td>
<td>1291</td>
</tr>
<tr>
<td>weekday_last</td>
<td>1292</td>
</tr>
<tr>
<td>year</td>
<td>1287</td>
</tr>
<tr>
<td>year_month</td>
<td>1297</td>
</tr>
<tr>
<td>year_month_day</td>
<td>1300</td>
</tr>
<tr>
<td>year_month_day_last</td>
<td>1302</td>
</tr>
<tr>
<td>year_month_weekday</td>
<td>1305</td>
</tr>
<tr>
<td>year_month_weekday_last</td>
<td>1307</td>
</tr>
<tr>
<td>zoned_time</td>
<td>1324</td>
</tr>
<tr>
<td>operator&gt;</td>
<td>1684</td>
</tr>
<tr>
<td>duration</td>
<td>1265</td>
</tr>
<tr>
<td>elements_view::iterator</td>
<td>1042</td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>1005</td>
</tr>
<tr>
<td>leap_second</td>
<td>1325</td>
</tr>
<tr>
<td>move_iterator</td>
<td>962</td>
</tr>
<tr>
<td>optional</td>
<td>610, 611</td>
</tr>
<tr>
<td>partial_ordering</td>
<td>540</td>
</tr>
<tr>
<td>queue</td>
<td>910</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>955</td>
</tr>
<tr>
<td>stack</td>
<td>915</td>
</tr>
<tr>
<td>strong_ordering</td>
<td>542</td>
</tr>
<tr>
<td>sys_time</td>
<td>1325</td>
</tr>
<tr>
<td>time_point</td>
<td>1270</td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>1017</td>
</tr>
<tr>
<td>type_index</td>
<td>736</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>657, 658</td>
</tr>
<tr>
<td>valarray</td>
<td>1220, 1221</td>
</tr>
<tr>
<td>variant</td>
<td>621</td>
</tr>
<tr>
<td>weak_ordering</td>
<td>541</td>
</tr>
<tr>
<td>operator&gt;=</td>
<td>1684</td>
</tr>
<tr>
<td>duration</td>
<td>1265</td>
</tr>
<tr>
<td>elements_view::iterator</td>
<td>1042</td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>1005</td>
</tr>
<tr>
<td>leap_second</td>
<td>1325, 1326</td>
</tr>
<tr>
<td>move_iterator</td>
<td>963</td>
</tr>
<tr>
<td>optional</td>
<td>610, 611</td>
</tr>
<tr>
<td>partial_ordering</td>
<td>540</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>956</td>
</tr>
<tr>
<td>stack</td>
<td>915</td>
</tr>
<tr>
<td>strong_ordering</td>
<td>542</td>
</tr>
<tr>
<td>sys_time</td>
<td>1325, 1326</td>
</tr>
<tr>
<td>time_point</td>
<td>1270</td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>1017</td>
</tr>
<tr>
<td>type_index</td>
<td>736</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>657, 658</td>
</tr>
<tr>
<td>valarray</td>
<td>1220, 1221</td>
</tr>
<tr>
<td>variant</td>
<td>621</td>
</tr>
<tr>
<td>weak_ordering</td>
<td>541</td>
</tr>
<tr>
<td>operator&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>basic_istream</td>
<td>1407–1409, 1413</td>
</tr>
<tr>
<td>basic_string</td>
<td>788</td>
</tr>
<tr>
<td>bitset</td>
<td>632, 633</td>
</tr>
<tr>
<td>byte</td>
<td>509</td>
</tr>
<tr>
<td>complex</td>
<td>1167</td>
</tr>
<tr>
<td>path</td>
<td>1477</td>
</tr>
<tr>
<td>valarray</td>
<td>1219, 1220</td>
</tr>
<tr>
<td>operator&gt;&gt;=</td>
<td></td>
</tr>
<tr>
<td>bitset</td>
<td>631</td>
</tr>
<tr>
<td>byte</td>
<td>509</td>
</tr>
<tr>
<td>gslice_array</td>
<td>1226</td>
</tr>
<tr>
<td>indirect_array</td>
<td>1228</td>
</tr>
<tr>
<td>mask_array</td>
<td>1227</td>
</tr>
<tr>
<td>slice_array</td>
<td>1223</td>
</tr>
<tr>
<td>valarray</td>
<td>1218</td>
</tr>
<tr>
<td>operator[]</td>
<td></td>
</tr>
<tr>
<td>basic_string</td>
<td>777</td>
</tr>
<tr>
<td>basic_string_view</td>
<td>796</td>
</tr>
<tr>
<td>bitset</td>
<td>632, 633</td>
</tr>
<tr>
<td>counted_iterator</td>
<td>970</td>
</tr>
<tr>
<td>indirect_array</td>
<td>1227</td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>1004</td>
</tr>
<tr>
<td>map</td>
<td>874</td>
</tr>
<tr>
<td>mask_array</td>
<td>1226</td>
</tr>
<tr>
<td>match_results</td>
<td>1525</td>
</tr>
<tr>
<td>move_iterator</td>
<td>961</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>954</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>663</td>
</tr>
<tr>
<td>span</td>
<td>920</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>656</td>
</tr>
<tr>
<td>unordered_map</td>
<td>892</td>
</tr>
<tr>
<td>valarray</td>
<td>1216, 1217</td>
</tr>
<tr>
<td>weekday</td>
<td>1289</td>
</tr>
<tr>
<td>operator%</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td>1264</td>
</tr>
<tr>
<td>valarray</td>
<td>1219, 1220</td>
</tr>
<tr>
<td>operator%=</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td>1263</td>
</tr>
<tr>
<td>gslice_array</td>
<td>1226</td>
</tr>
<tr>
<td>indirect_array</td>
<td>1228</td>
</tr>
<tr>
<td>mask_array</td>
<td>1227</td>
</tr>
<tr>
<td>slice_array</td>
<td>1223</td>
</tr>
<tr>
<td>valarray</td>
<td>1218</td>
</tr>
<tr>
<td>operator&amp;</td>
<td></td>
</tr>
<tr>
<td>bitset</td>
<td>633</td>
</tr>
<tr>
<td>byte</td>
<td>509</td>
</tr>
<tr>
<td>valarray</td>
<td>1219, 1220</td>
</tr>
<tr>
<td>operator&amp;=</td>
<td></td>
</tr>
<tr>
<td>atomic&lt;integral&gt;</td>
<td>1559</td>
</tr>
<tr>
<td>atomic_ref&lt;integral&gt;</td>
<td>1550</td>
</tr>
<tr>
<td>bitset</td>
<td>630</td>
</tr>
<tr>
<td>byte</td>
<td>509</td>
</tr>
<tr>
<td>gslice_array</td>
<td>1226</td>
</tr>
<tr>
<td>indirect_array</td>
<td>1228</td>
</tr>
<tr>
<td>mask_array</td>
<td>1227</td>
</tr>
<tr>
<td>slice_array</td>
<td>1223</td>
</tr>
</tbody>
</table>
valarray, 1218
operator&&
valarray, 1220, 1221
operator~
bitset, 633
byte, 510
valarray, 1219, 1220
operator=
atomic<integral>, 1559
atomic_ref<integral>, 1550
bitset, 631
byte, 509
gslice_array, 1226
indirect_array, 1228
mask_array, 1227
slice_array, 1223
valarray, 1218
operator~
bitset, 631
byte, 510
valarray, 1217
operator--
counted_iterator, 971
operator|
bitset, 633
byte, 509
valarray, 1219, 1220
operator|= atomic<integral>, 1559
atomic_ref<integral>, 1550
bitset, 630
byte, 509
gslice_array, 1226
indirect_array, 1228
mask_array, 1227
slice_array, 1223
valarray, 1218
operator||
valarray, 1220, 1221
optimize syntax_option_type, 1512
optional, 601
constructor, 602–604
destructor, 604
emplace, 607
has_value, 608
hash, 612
operator bool, 608
operator!, 610, 611
operator*, 608
operator->, 608
operator<, 610, 611
operator<=, 610, 611
operator<>, 610, 611
operator=, 604–606
operator==, 609, 610
operator>, 610, 611
operator>=, 610, 611
reset, 609
swap, 607, 611
value, 608, 609
value_or, 609
value_type, 601
options recursive_directory_iterator, 1489
synchronized_pool_resource, 679
unsynchronized_pool_resource, 679
ostream, 1376, 1403
ostream_iterator, 975
constructor, 975
operator*, 975
operator++, 976
operator=, 975
ostreambuf_iterator, 977, 1376
constructor, 978
failed, 978
operator*, 978
operator++, 978
operator=, 978
ostreamstringstream, 1376, 1427
ostringstream, 1376, 1490
 SUCCESS, 1376, 1453
out
basic_format_context, 754
codecvt, 1349
format_to_n_result, 741
out_of_range, 566, 567, 630–632
constructor, 568
outer_allocator
scoped_allocator_adaptor, 684
outer_allocator_type
scoped_allocator_adaptor, 681
output_iterator
output_iterator_tag, 949
output_range, 992
overflow
basic_filebuf, 1446
basic_stringbuf, 1402
basic_stringbuf, 1432
strstreambuf, 1687
overflow_error, 506, 568, 630, 632
constructor, 568, 569
owner_before
shared_ptr, 663
weak_ptr, 670
owner_less
operator(), 671
owns_lock
shared_lock, 1602
unique_lock, 1598
bernoulli_distribution, 1195
binomial_distribution, 1196
gaussian_distribution, 1197
negative_binomial_distribution, 1197
packaged_task, 1627
constructor, 1628
destructor, 1628
get_future, 1628
make_ready_at_thread_exit, 1629
operator(), 1629
operator=, 1628
reset, 1629
swap, 1628, 1629
valid, 1628
pair, 586, 594–596
constructor, 586, 587
get, 589, 590
operator<=>, 589
operator=, 588
operator==, 588
swap, 588, 589
par, 738
execution, 738
par_unseq, 738
execution, 738
param
seed_seq, 1192
parent_path
path, 1474
parse, 1330–1333
partial_order, 545
partial_ordering, 539
constructor, 539
less, 539
operator<, 540
operator<=, 540
operator<=>, 540
operator==, 540
operator>, 540
operator>>, 540
unordered, 539
partial_sort, 1117
partial_sort_copy, 1117
partial_sum, 1148
partition, 1123
partition_copy, 1124
partition_point, 1125
path, 1463
append, 1471
assign, 1470
begin, 1477
c_str, 1472
clear, 1471
compare, 1473
concat, 1471
constructor, 1469
copy, 1491
directory_entry, 1484
empty, 1475
end, 1477
extension, 1474
filename, 1474
generic_string, 1473
generic_u16string, 1473
generic_u32string, 1473
generic_u8string, 1473
generic_wstring, 1473
has_extension, 1475
has_filename, 1475
has_parent_path, 1475
has_relative_path, 1475
has_root_directory, 1475
has_root_name, 1475
has_root_path, 1475
has stem, 1475
hash_value, 1475
is_absolute, 1475
is_relative, 1475
iterator, 1476
lexically_normal, 1475
lexically_proximate, 1476
lexically_relative, 1476
make_preferred, 1471
native, 1472
operator string_type, 1472
operator+=, 1471
operator/, 1478
operator/=, 1470, 1471
operator<<, 1477
operator<=, 1477
operator==, 1470
operator>>, 1477
parent_path, 1474
preferred_separator, 1465
relative_path, 1474
remove, 1500
remove_filename, 1472
replace_extension, 1472
replace_filename, 1472
root_directory, 1474
root_name, 1474
root_path, 1474
stem, 1474
string, 1472, 1473
swap, 1472, 1477
u16string, 1473
u32string, 1473
u8string, 1473
value_type, 1465
wstring, 1473
path1
filesystem_error, 1478
path2
filesystem_error, 1478
pbackfail
  basic_filebuf, 1445
  basic_streambuf, 1402
  basic_stringbuf, 1432
  streambuf, 1688
pbase
  basic_streambuf, 1400
pbump
  basic_streambuf, 1400
pcount
  ostrstream, 1691
  stream, 1692
  streambuf, 1687
peek
  basic_istream, 1412
perm_options, 1481
permissions, 1499
  file_status, 1482
perms, 1481
permutable, 949
perror, 1503
piecewise_constant_distribution, 1207
  constructor, 1207, 1208
  densities, 1208
  intervals, 1208
  result_type, 1207
piecewise_construct, 590
piecewise_construct_t, 590
piecewise_linear_distribution, 1208
  constructor, 1209
  densities, 1210
  intervals, 1209
  result_type, 1208
placeholders, 701
plus, 690
  operator(), 690
plus<>, 690
  operator(), 690
pmr::string
  hash, 791
pmr::u16string
  hash, 791
pmr::u32string
  hash, 791
pmr::wstring
  hash, 791
pointer
  allocator_traits, 646
  basic_string, 768
  basic_string_view, 792
  iterator_traits, 932
  scoped_allocator_adaptor, 681
pointer_traits
  preferred, 634
  relaxed, 634
  strict, 634
pointer_to
  pointer_traits, 642
pointer_to_binary_function
  zombie, 499
pointer_to_unary_function
  zombie, 499
pointer_traits, 641
  difference_type, 641
  element_type, 641
  pointer_to, 642
  rebind, 641
  to_address, 642
poisson_distribution, 1197, 1198
  constructor, 1198
  mean, 1198
  result_type, 1198
polar
  complex, 1168
polymorphic_allocator, 674
  allocate, 675
  allocate_bytes, 675
  allocate_object, 675
  construct, 676
  constructor, 675
  deallocate, 675
  deallocate_bytes, 675
  deallocate_object, 675
  destroy, 676
  new_object, 675, 676
  operator==, 676
  resource, 676
  select_on_container_copy_construction, 676
  value_type, 674
pool_options, 677
  largest_required_pool_block, 678
  max_blocks_per_chunk, 678
pop
  forward_list, 852
  priority_queue, 913
  recursive_directory_iterator, 1490
pop_back
  basic_string, 781
pop_heap, 1133
popcount, 1173
position
  match_results, 1525
positive_sign
  moneypunct, 1371
pow, 1170, 1229
  complex, 1169
  valarray, 1221
powl, 1229
powf, 1229
pptr
  basic_streambuf, 1400
precision
  ios_base, 1342, 1385
pred
  drop_while_view, 1025
  filter_view, 1011
Index of library names
puts, 1503
putc, 802
putwchar, 802
pword
  ios_base, 1387
Q
qsort, 507, 1160
queue, 909
  operator<, 910
  operator<=, 910
  operator<>, 911
  operator==, 910
  operator>=, 910
  swap, 911
quick_exit, 487, 507, 522
quiet_NaN
  numeric_limits, 517
quoted, 1426
R
radix
  numeric_limits, 515
raise, 551
rand, 507, 1210
discouraged, 1210
RAND_MAX, 507
random_access_iterator, 941
random_access_iterator_tag, 949
random_access_range, 992
random_device, 1190
  constructor, 1190
  entropy, 1191
  operator(), 1191
  result_type, 1190
random_shuffle
  zombie, 499
range, 990
range_error, 566, 568
  constructor, 568
rank, 723
rank_v, 714
ranlux24, 1190
ranlux24_base, 1190
ranlux48, 1190
ranlux48_base, 1190
ratio, 733, 734
  ratio_equal, 735
  ratio_equal_v, 733
  ratio_greater, 735
  ratio_greater_equal, 735
  ratio_greater_equal_v, 733
  ratio_greater_v, 733
  ratio_less, 735
  ratio_less_equal, 735
  ratio_less_equal_v, 733
  ratio_less_v, 733
  ratio_not_equal, 735
  ratio_not_equal_v, 733
raw_storage_iterator
  zombie, 499
rbegin, 987
  basic_string, 776
  basic_string_view, 795
  span, 921
rbegin(C&), 979
rbegin(initializer_list<E>), 979
rbegin(T (&array)[N]), 979
rdbuf
  basic_fstream, 1452
  basic_ifstream, 1449
  basic_ios, 1390
  basic_ifstream, 1435
  basic_ofstream, 1450
  basic_ostringstream, 1438
  basic_stringstream, 1440
  iostreamstream, 1690
  ostrstream, 1691
  strstream, 1692
    wbuffer_convert, 1701
rdstate
  basic_ios, 1392
read
  basic_istream, 1412
  read_symlink, 1499
readsome
  basic_istream, 1412
ready
  match_results, 1525
real, 1170
  complex, 1166, 1168
realloc, 507, 649, 1680
rebind
  pointer_traits, 641
rebind_alloc
  allocator_traits, 647
recursion_pending
  recursive_directory_iterator, 1490
recursive_directory_iterator, 1488
begin, 1490
  constructor, 1489
depth, 1490
disable_recursion_pending, 1490
end, 1490
increment, 1490
operator++, 1490
operator=, 1489
options, 1489
pop, 1490
recursion_pending, 1490
recursive_mutex, 1588
recursive_timed_mutex, 1590
reduce, 1145, 1146
ref
  reference_wrapper, 690
ref_view, 1009, 1010
Index of library names 1822
Index of library names

unordered associative containers, 838
reinterpret_pointer_cast
shared_ptr, 668
rel_ops, 1684
relation, 564
relative, 1499
relative_path
path, 1474
relaxed
memory_order, 1543
pointer_safety, 634
release
counting_semaphore, 1612
memory_order, 1543
monotonic_buffer_resource, 680
shared_lock, 1602
synchronized_pool_resource, 679
unique_lock, 1598
unique_ptr, 654
unsynchronized_pool_resource, 679
reload_tzdb, 1315
remainder, 1229
remainderf, 1229
remainderl, 1229
remote_version, 1316
remove, 1107, 1503
forward_list, 854
list, 860
path, 1500
remove_all, 1500
remove_all_extents, 727
remove_all_extents_t, 711
remove_const, 726
remove_const_t, 710
remove_copy, 1108
remove_copy_if, 1108
remove_cv, 726
remove_cv_t, 710
remove_cvref, 728
remove_cvref_t, 712
remove_extent, 727
remove_extent_t, 711
remove_filename
path, 1472
remove_if, 1107
forward_list, 854
remove_pointer, 727
remove_pointer_t, 711
remove_prefix
basic_string_view, 796
remove_reference, 726
remove_reference_t, 711
remove_suffix
basic_string_view, 796
remove_volatile, 726
remove_volatile_t, 710
remquo, 1229
remquof, 1229
remquol, 1229

reference
basic_string, 768
basic_string_view, 792
iterator_traits, 932
reference_wrapper, 688
constructor, 689
cref, 690
get, 689
operator T&, 689
operator(), 690
operator=, 689
ref, 690
refresh
directory_entry, 1484
regex, 1508
regex_constants, 1512
error_type, 1513, 1514
match_flag_type, 1513
syntax_option_type, 1512
regex_error, 1514, 1517, 1537
constructor, 1514
regex_iterator, 1531
constructor, 1532
increment, 1533
operator*, 1533
operator++, 1533
operator->, 1533
operator==, 1532
regex_match, 1527–1529
regex_replace, 1530, 1531
regex_search, 1529, 1530
regex_token_iterator, 1533
constructor, 1535
end-of-sequence, 1534
operator*, 1536
operator++, 1536
operator->, 1536
operator==, 1534, 1536
regex_traits, 1514
char_class_type, 1515
isctype, 1516
length, 1515
lookup_classname, 1516
lookup_collatename, 1515
transform, 1515
transform_primary, 1515
translate, 1515
translate_nocase, 1515
value, 1516
register_callback
ios_base, 1387
regular, 564
regular_expression_traits
isctype, 1537
lookup_classname, 1537
lookup_collatename, 1537
transform_primary, 1537
regular_invocable, 564
rehash

unrecognized
Index of library names

unordered associative containers, 838
reinterpret_pointer_cast
shared_ptr, 668
rel_ops, 1684
relation, 564
relative, 1499
relative_path
path, 1474
relaxed
memory_order, 1543
pointer_safety, 634
release
counting_semaphore, 1612
memory_order, 1543
monotonic_buffer_resource, 680
shared_lock, 1602
synchronized_pool_resource, 679
unique_lock, 1598
unique_ptr, 654
unsynchronized_pool_resource, 679
reload_tzdb, 1315
remainder, 1229
remainderf, 1229
remainderl, 1229
remote_version, 1316
remove, 1107, 1503
forward_list, 854
list, 860
path, 1500
remove_all, 1500
remove_all_extents, 727
remove_all_extents_t, 711
remove_const, 726
remove_const_t, 710
remove_copy, 1108
remove_copy_if, 1108
remove_cv, 726
remove_cv_t, 710
remove_cvref, 728
remove_cvref_t, 712
remove_extent, 727
remove_extent_t, 711
remove_filename
path, 1472
remove_if, 1107
forward_list, 854
remove_pointer, 727
remove_pointer_t, 711
remove_prefix
basic_string_view, 796
remove_reference, 726
remove_reference_t, 711
remove_suffix
basic_string_view, 796
remove_volatile, 726
remove_volatile_t, 710
remquo, 1229
remquof, 1229
remquol, 1229

unrecognized
Index of library names
rename, 1500, 1503
rend, 987
   basic_string, 776
   basic_string_view, 795
span, 921
rend(C&), 979
rend(initializer_list<E>), 979
rend(T (&array)[N]), 979
rep
   system_clock, 1271
replace, 1104
   basic_string, 781–783
replace_copy, 1105
replace_copy_if, 1105
replace_extension
   path, 1472
replace_filename
   directory_entry, 1484
   path, 1472
replace_if, 1104
request_stop
   jthread, 1585
stop_source, 1577
required_alignment
   atomic_ref, 1547
   atomic_ref<floating-point>, 1547
   atomic_ref<integral>, 1547
   atomic_ref<T>, 1547
reserve
   basic_string, 776, 1696
   unordered associative containers, 839
   vector, 864
reset
   any, 626
   bitset, 631
   optional, 609
   packaged_task, 1629
   shared_ptr, 662, 663
   unique_ptr, 654, 656
   weak_ptr, 670
resetiosflags, 1423
resize
   basic_string, 776
   deque, 848
   forward_list, 853
   list, 858
   valarray, 1219
   vector, 864, 865
resize_file, 1500
resource
   polymorphic_allocator, 676
result
   local_info, 1318
result_of
   zombie, 499
result_of_t
   zombie, 499
result_type
   bernoulli_distribution, 1195
   binomial_distribution, 1195
   cauchy_distribution, 1203
   chi_squared_distribution, 1203
   discard_block_engine, 1187
   discrete_distribution, 1205
   exponential_distribution, 1198
   extreme_value_distribution, 1200
   fisher_distribution, 1204
   function, 702
   gamma_distribution, 1199
   geometric_distribution, 1196
   independent_bits_engine, 1188
   linear_congruential_engine, 1183
   lognormal_distribution, 1202
   mersenne_twister_engine, 1184
   negative_binomial_distribution, 1197
   normal_distribution, 1201
   piecewise_constant_distribution, 1207
   piecewise_linear_distribution, 1208
   poisson_distribution, 1198
   random_device, 1190
   seed_seq, 1191
   shuffle_order_engine, 1188
   student_t_distribution, 1205
   subtract_with_carry_engine, 1185
   uniform_int_distribution, 1193
   uniform_real_distribution, 1194
   weibull_distribution, 1200
resume
   coroutine_handle, 548
   coroutine_handle<noop_coroutine_-promise>,
   549
rethrow_exception, 536
rethrow_if_nested
   nested_exception, 537
rethrow_nested
   nested_exception, 537
return_temporary_buffer
   zombie, 499
reverse, 1111
   forward_list, 855
   list, 861
   reverse_copy, 1111
   reverse_iterator, 953
      base, 954
      basic_string, 768
      basic_string_view, 792
      constructor, 954
      iter_move, 956
      iter_swap, 956
      make_reverse_iterator non-member
         function, 956
      operator!=, 955
      operator*, 954
      operator+, 954, 956
      operator++, 954, 955
      operator+=, 955
      operator-, 954, 956

Index of library names
1824
operator-, 955  
operator->, 954  
operator--, 955  
operator<, 955  
operator<=, 956  
operator=, 954  
operator==, 955  
operator>, 955  
operator>=, 956  
operator[], 954  

reverse_view, 1037  
base, 1037  
begin, 1038  
constructor, 1038  
end, 1038  
size, 1037  

REWIND, 1503  
rfind  
   basic_string, 784  
   basic_string_view, 798  
riemann_zeta, 1242  
riemann_zetal, 1242  
right, 1394  
rint, 1229  
rintf, 1229  
rintl, 1229  
root_directory  
   path, 1474  
root_name  
   path, 1474  
root_path  
   path, 1474  
rotate, 1112  
rotate_copy, 1112  
rotl, 1172  
rotr, 1172  
round, 1229  
   duration, 1266  
   time_point, 1270  
round_error  
   numeric_limits, 515  
   round_indeterminate, 513  
   round_style  
      numeric_limits, 517  
   round_to_nearest, 513  
   round_toward_infinity, 513  
   round_toward_neg_infinity, 513  
   round_toward_zero, 513  
roundf, 1229  
roundl, 1229  
runtime_error, 566, 568  
   constructor, 568  

S  

same_as, 556  
sample, 1113  
save  
   sys_info, 1317  
sbumpc  
   basic_streambuf, 1398  
scalbln, 1229  
scalblnf, 1229  
scalblnl, 1229  
scalbn, 1229  
scalbnf, 1229  
scalbnl, 1229  
scan_is  
   ctype, 1344  
      ctype<char>, 1347  
scan_not  
   ctype, 1344  
      ctype<char>, 1347  
scanf, 1503  
SCHAR_MAX, 519  
SCHAR_MIN, 519  
scientific, 1394  
   chars_format, 739  
SCNdFASTN, 1504  
SCNdLEANSTN, 1504  
SCNdMAX, 1504  
SCNdN, 1504  
SCNdPTR, 1504  
SCNdFASTN, 1504  
SCNdLEANSTN, 1504  
SCNdMAX, 1504  
SCNdN, 1504  
SCNdPTR, 1504  
SCNuFASTN, 1504  
SCNuLEANSTN, 1504  
SCNuMAX, 1504  
SCNuN, 1504  
SCNuPTR, 1504  
SCNuFASTN, 1504  
SCNuLEANSTN, 1504  
SCNuMAX, 1504  
SCNuN, 1504  
SCNuPTR, 1504  
SCNxFASTN, 1504  
SCNxLEANSTN, 1504  
SCNxMAX, 1504  
SCNxN, 1504  
SCNxPTR, 1504  
scoped_allocator_adaptor, 681  
   allocate, 684  
   const_pointer, 681  
   const_void_pointer, 681  
   construct, 684  
   constructor, 683  
   deallocate, 684  
   destroy, 684  
   difference_type, 681  
   inner_allocator, 684  
   inner_allocator_type, 683  

lognormal_distribution, 1202
atomic_compare_exchange_strong, 1696
atomic_compare_exchange_strong_explicit, 1696
atomic_compare_exchange_weak, 1696
atomic_compare_exchange_weak_explicit, 1696
atomic_exchange, 1695
atomic_exchange_explicit, 1695
atomic_is_lock_free, 1695
atomic_load, 1695
atomic_load_explicit, 1695
atomic_store, 1695
atomic_store_explicit, 1695
const_pointer_cast, 668
constructor, 660–662
destructor, 662
dynamic_pointer_cast, 667
g, 663
get, 663
get_deleter, 668
hash, 672
operator bool, 663
operator*, 663
operator->, 663
operator<<, 668
operator<<=, 667
operator=, 662
operator==, 667
operator[], 663
owner_before, 663
reinterpret_pointer_cast, 668
reset, 662, 663
static_pointer_cast, 667
swap, 662, 667
use_count, 663
shared_timed_mutex, 1593
shift
tavalarray, 1219
shift_left, 1114
shift_right, 1114
showbase, 1393
showmanyc
basic_filebuf, 1445
basic_streambuf, 1401, 1445
showpoint, 1393
showpos, 1393
shrink_to_fit
basic_string, 776
deque, 848
vector, 864
SHRT_MAX, 519
SHRT_MIN, 519
shuffle, 1114
shuffle_order_engine, 1188
constructor, 1189
result_type, 1188
sig_atomic_t, 551
SIG_DFL, 551
SIG_ERR, 551
Index of library names
Index of library names

SIG_IGN, 551
SIGABRT, 551
SIGFPE, 551
SIGILL, 551
SIGINT, 551
signal, 551
signaling_NaN
   numeric_limits, 517
signbit, 1229
signed_integral, 558
SIGSEGV, 551
SIGTERM, 551
sin, 1229
   complex, 1169
   valarray, 1221
sinf, 1229
single_view, 999
   begin, 1000
   constructor, 999, 1000
   data, 1000
   end, 1000
   size, 1000
sinh, 1229
   complex, 1169
   valarray, 1221
sinhf, 1229
sinhl, 1229
sinl, 1229
size, 988
   array, 843, 844
   basic_string, 776
   basic_string_view, 795
   bitset, 632
   common_view, 1036
   drop_view, 1023
   elements_view, 1039
   format_to_n_result, 741
gslice, 1225
initializer_list, 538
iota_view, 1002
match_results, 1525
reverse_view, 1037
seed_seq, 1192
single_view, 1000
slice, 1222
span, 920
subrange, 997
take_view, 1019
transform_view, 1013
valarray, 1218
size(C& c), 979
size(T (&array)[N]), 979
size_bytes
   span, 920
size_t, 130, 506, 507, 801–803, 1334, 1503
size_type
   allocator, 648
   allocator_traits, 647
   basic_string, 768
   basic_string_view, 792
   scoped_allocator_adaptor, 681
sized_range, 991
sized_sentinel_for, 939
sleep
   this_thread, 1585
sleep_until
   this_thread, 1585
slice, 1222
   constructor, 1222
   size, 1222
   start, 1222
   stride, 1222
slice_array, 1223
   operator**, 1223
   operator**, 1223
   operator=, 1223
   operator/=, 1223
   operator<<, 1223
   operator>>, 1223
   operator|, 1223
   operator^, 1223
   operator&=, 1223
   operator%=, 1223
   operator%=, 1223
   value_type, 1223
snextc
   basic_string_view, 1398
snprintf, 1503
sort, 1115
   forward_list, 855
   list, 861
sortable, 949
source_location, 531
space, 1500
span, 916
   back, 920
   begin, 921
   constructor, 917–919
   data, 920
   deduction guide, 919
   empty, 920
   end, 921
   first, 919, 920
   front, 920
   iterator, 921
   last, 919, 920
   operator=, 919
   rbegin, 921
   rend, 921
   size, 920
   size_bytes, 920
   subspan, 919, 920
sph_bessel, 1243
sph_besself, 1243
sph_bessell, 1243
sph_legendre, 1243
Index of library names
store
  atomic, 1554
  atomic<floating-point>, 1554
  atomic<integral>, 1554
  atomic<T*>, 1554
  atomic<shared_ptr<T>>, 1564
  atomic_ref, 1547
  atomic_ref<floating-point>, 1547
  atomic_ref<integral>, 1547
  atomic_ref<T*>, 1547

stossc
  zombie, 499

stoul, 789, 790
stoull, 789, 790

str
  basic_istringstream, 1435, 1436
  basic_ostringstream, 1438
  basic_stringbuf, 1431
  basic_stringstream, 1440, 1441
  istrstream, 1690
  match_results, 1525
  ostringstream, 1691
  strstream, 1692
  strstreambuf, 1687
  sub_match, 1522
strcat, 801
strchr, 801
strcspn, 801
strcoll, 801
strcpyp, 801
strncpy, 801

streambuf, 1376, 1395
streamoff, 1380, 1388
streampos, 1376
streamsize, 1381

strerror, 801
strerror, 801

strict
  pointer_safety, 634
  strict_weak_order, 565

stride
  gslice, 1225
  slice, 1222

string, 767
  hash, 791
  operator"", 791
  path, 1472, 1473

string_view
  hash, 800
  operator"", 800
stringbuf, 1376, 1427
stringstream, 1376, 1427
strlen, 801, 1687, 1691
strncat, 801
strncpy, 801
strong_order, 544
strong_ordering, 541

equal, 541
equivalent, 541
greater, 541
less, 541
operator partial_ordering, 542
operator weak_ordering, 542
operator<, 542
operator<=, 542
operator=, 542
operator=, 542
operator>=, 542
strpbrk, 801
strrchr, 801
strspn, 801
strstr, 801

strftime, 1334, 1367

strtod, 507
strtof, 507
strtoimax, 1504
strtok, 801

strtol, 507
strtold, 507
strtoll, 507
strtoull, 507
strxfrm, 801

student_t_distribution, 1205
  constructor, 1205
  mean, 1205
  result_type, 1205

sub_match, 1521
  compare, 1522
  constructor, 1521
  length, 1521
  operator basic_string, 1522
  operator<, 1523
  operator=, 1522
  operator=, 1522
  str, 1522

Index of library names 1830
subrange, 994
  advance, 997
begin, 997
constructor, 996, 997
empty, 997
end, 997
get, 998
next, 997
operator PairLike, 997
prev, 997
size, 997
subseconds
  hh_mm_ss, 1312
subspan
  span, 919, 920
substr
  basic_string, 785
  basic_string_view, 796
subtract_with_carry_engine, 1185
  constructor, 1186
  result_type, 1185
suffix
  match_results, 1526
sum
  valarray, 1219
sungetc
  basic_streambuf, 1399
suspend_always, 550
  await_ready, 550
  await_resume, 550
  await_suspend, 550
suspend_never, 550
  await_ready, 550
  await_resume, 550
  await_suspend, 550
swap, 559, 582, 583
  any, 626, 627
array, 844
  basic_filebuf, 1443
  basic_fstream, 1452
  basic_ifstream, 1448
  basic_ios, 1392
  basic_iostream, 1414
  basic_istream, 1406
  basic_iostreamstream, 1435
  basic_ofstream, 1450
  basic_ostringstream, 1438
  basic_regex, 1521
  basic_streambuf, 1399
  basic_string, 783, 788
  basic_string_view, 796
  basic_stringbuf, 1430
  basic_stringstream, 1440
  basic_syncbuf, 1455, 1456
function, 704, 705
jthread, 1584, 1585
match_results, 1527
optional, 607, 611
packaged_task, 1628, 1629
pair, 588, 589
path, 1472, 1477
priority_queue, 913
promise, 1620, 1621
queue, 911
shared_lock, 1602
shared_ptr, 662, 667
stack, 915
stop_source, 1577
stop_token, 1575
thread, 1581, 1582
tuple, 596, 599
unique_lock, 1598
unique_ptr, 654
valarray, 1218, 1222
variant, 619, 622
vector, 864
vector<bool>, 868
weak_ptr, 670
swap(unique_ptr&, unique_ptr&), 657
swap_ranges, 1102
swappable, 559
swappable_with, 559
swprintf, 802
swscanf, 802
symlink_status, 1502
directory_entry, 1486
sync
  basic_filebuf, 1447
  basic_istream, 1413
  basic_streambuf, 1400
  basic_syncbuf, 1455
sync_with_stdio
  ios_base, 1386
synchronized_pool_resource, 677
  constructor, 678
destructor, 679
do_allocate, 679
do_deallocate, 679
do_is_equal, 679
options, 679
release, 679
upstream_resource, 679
syntax_option_type, 1512
awk, 1512
basic, 1512
collate, 1512, 1538
ECMAScript, 1512
egrep, 1512
extended, 1512
grep, 1512
icase, 1512
multiline, 1512
nosubs, 1512
optimize, 1512
regex_constants, 1512
sys_days, 1245
Index of library names
three_way_comparable, 543
three_way_comparable_with, 543, 544
throw_with_nested
    nested_exception, 537
tie, 596
    basic_ios, 1390
tuple, 596
time, 1334
time_get, 1362
do_date_order, 1363
do_get, 1365
do_get_date, 1364
do_get_monthname, 1365
do_get_time, 1364
do_get_weekday, 1365
do_get_year, 1365
get, 1363
get_date, 1363
get_monthname, 1363
get_time, 1363
get_weekday, 1363
get_year, 1363
time_get_byname, 1365
time_point, 1268
ceil, 1270
constructor, 1268
date_order, 1363
floor, 1270
date_order, 1364
date, 1269
max, 1269
min, 1269
operator+, 1269
operator++, 1269
operator=, 1269
operator-, 1269
operator--, 1269
operator=, 1269
operator<, 1270
operator<=, 1270
operator==, 1270
operator>, 1270
operator>=, 1270
round, 1270
time_point_cast, 1270
time_since_epoch, 1269
time_zone_link, 1326
name, 1326
operator<>, 1326
operator==, 1326
target, 1326
timed_mutex, 1589
timespec, 1334
timespec_get, 1334
tinyness_before
    numeric_limits, 517
tm, 802, 1334
TMP_MAX, 1503
tmpfile, 1503
tmpnam, 1503
to_address, 642
    pointer_traits, 642
to_array, 845
to_bytes
    wstring_convert, 1700
to_chars, 740
to_chars_result, 739
ec, 739
ptr, 739
to_duration
    hh_mm_ss, 1312
to_integer
    byte, 510
to_local
time_zone, 1319
to_string, 790
    bitset, 632
to_sys
    time_zone, 1319
    utc_clock, 1273
to_time_t
    system_clock, 1271
to_ullong
    bitset, 632
to_ullong
    bitset, 632
to_utc
gps_clock, 1276
tai_clock, 1275
to_wstring, 790
tolower, 800, 1342
cctype, 1344
cctype<char>, 1347
totally_ordered, 563
totally_ordered_with, 563
toupper, 800, 1342
cctype, 1344
cctype<char>, 1347
towctrans, 801
towlower, 801
towupper, 801
traits_type
    basic_string, 768
Index of library names

basic_string_view, 792
transform, 1103
collate, 1361
regex_traits, 1515
transform_exclusive_scan, 1150
transform_inclusive_scan, 1151
transform_primary
regex_traits, 1515
transform_reduce, 1146, 1147
transform_view, 1013
  base, 1013
  begin, 1014
  constructor, 1014
  end, 1014, 1015
  size, 1013
transform_view::iterator, 1015
  base, 1016
  constructor, 1016
  iter_swap, 1018
  iterator, 1016
  operator+, 1018
  operator++, 1017
  operator+=, 1017
  operator-, 1018
  operator=, 1017
  operator-=, 1017
  operator==, 1017
  operator>==, 1017
transform_view::sentinel, 1018
  base, 1019
  constructor, 1018
  operator=, 1019
  operator==, 1019
translate
  regex_traits, 1515
translate_nocase
regex_traits, 1515
traps
  numeric_limits, 517
treat_asFloating_point, 1259
treat_asFloating_point_v, 1245
tuple
  constructor, 593, 594
  forward_as_tuple, 596
  make_tuple, 596
  operator<=>, 599
  operator=, 595, 596
  operator==, 599
  swap, 596, 599
  tie, 596
tuple_cat, 597
tuple_element, 589, 597, 598, 845
tuple_element_t, 591
tuple_size, 589, 597, 598, 845
  in general, 597
tuple_size_v, 591
type
  any, 627
  file_status, 1482
type_identity, 728
  type_identity_t, 712
type_index, 736
  constructor, 736
  hash, 737
  hash_code, 736
  name, 736
  operator<, 736
  operator<=, 736
  operator<>, 736
  operator==, 736
  operator>, 736
  operator>=, 736
type_info, 122, 530
  before, 530
  hash_code, 530
  name, 530
  operator==, 530
tzdb, 1313
  current_zone, 1314
  locate_zone, 1314
tzdb_list, 1314
  begin, 1315
cbegin, 1315
cend, 1315
  end, 1315
unordered_map, 893
try_lock, 1602
  shared_lock, 1601
  unique_lock, 1597
try_lock_for
  shared_lock, 1601
  unique_lock, 1598
try_lock_until
  shared_lock, 1601
  unique_lock, 1598
try_to_lock, 1594
try_to_lock_t, 1594
try_wait
  latch, 1613
erase_after, 1314
front, 1314
U
u16streampos, 1376
u16string, 767
hash, 791
operator"s, 791
path, 1473
u16string_view
hash, 800
operator"sv, 800
u32streampos, 1376
u32string, 767
hash, 791
operator"s, 791
path, 1473
u32string_view
hash, 800
operator"sv, 800
u8path, 1702
u8string, 767
operator"s, 791
path, 1473
u8string_view
hash, 800
operator"sv, 800
UCHAR_MAX, 519
uflow
basic_filebuf, 1445
basic_streambuf, 1401
uint16_t, 520
uint32_t, 520
uint64_t, 520
uint8_t, 520
uint_fast16_t, 520
uint_fast32_t, 520
uint_fast64_t, 520
uint_fast8_t, 520
uint_least16_t, 520
uint_least32_t, 520
uint_least64_t, 520
uint_least8_t, 520
UINT_MAX, 519
uintmax_t, 520
uintptr_t, 520
ULLONG_MAX, 519
ULONG_MAX, 519
unary_function
zombie, 499
unary_negate
zombie, 499
uncaught_exception
zombie, 499
uncaught_exceptions, 460, 535
undeclare_no_pointers, 643
undeclare_reachable, 642
underflow
basic_filebuf, 1445
basic_streambuf, 1401
basic_stringbuf, 1432
strstreambuf, 1688
underflow_error, 566, 569
constructor, 569
underlying_type, 728
underlying_type_t, 712
unexpected
zombie, 499
unexpected_handler
zombie, 499
unget
basic_istream, 1412
ungetc, 1503
ungetwc, 802
uniform_int_distribution, 1193
a, 1194
b, 1194
constructor, 1194
result_type, 1193
uniform_random_bit_generator, 1177
uniform_real_distribution, 1194
a, 1194
b, 1195
constructor, 1194
result_type, 1194
uninitialized_construct_using_allocator, 646
uninitialized_copy, 1156
uninitialized_copy_n, 1157
uninitialized_default_construct, 1155
uninitialized_default_construct_n, 1155
uninitialized_fill, 1158
uninitialized_fill_n, 1158, 1159
uninitialized_move, 1157
uninitialized_move_n, 1158
uninitialized_value_construct, 1156
uninitialized_value_construct_n, 1156
unique, 1109
forward_list, 854
list, 860
local_info, 1318
unique_copy, 1110
unique_lock, 1595
constructor, 1596, 1597
destructor, 1597
lock, 1597
mutex, 1599
operator bool, 1599
operator=, 1597
owns_lock, 1598
release, 1598
swap, 1598
try_lock, 1597
try_lock_for, 1598
try_lock_until, 1598
unlock, 1598
unique_ptr, 650, 654, 662
Index of library names 1835
constructor, 651, 652, 655, 656
destructor, 653
get, 654
get_deleter, 654
hash, 672
operator bool, 654
operator*, 653
operator->, 654
operator<, 657, 658
operator<=, 657, 658
operator<=>, 657, 658
operator=, 653, 656
operator==, 657
operator>, 657, 658
operator>=, 657, 658
operator[], 654
release, 654
reset, 654, 656
swap, 654
unitbuf, 1393
unlock
    shared_lock, 1601
    unique_lock, 1598
unordered
    partial_ordering, 539
unordered_map, 886, 888
at, 892
begin, 838
bucket, 838
bucket_count, 837
bucket_size, 838
cbegin, 838
cend, 838
clear, 837
const_local_iterator, 830
constructor, 830, 892
contains, 837
count, 837
emplace, 833
emplace_hint, 833
depend, 838
equal_range, 837
erase, 836
erase_if, 894
extract, 836
find, 837
hash_function, 833
hasher, 830
insert, 834, 898
key_eq, 833
key_equal, 830
key_type, 830
load_factor, 838
local_iterator, 830
mapped_type, 830
max_bucket_count, 837
max_load_factor, 837
merge, 836
node_type, 830
operator[], 892
rehash, 838
reserve, 839
try_emplace, 893
value_type, 830
unordered_multimap, 886, 894
begin, 838
bucket, 838
bucket_count, 837
bucket_size, 838
cbegin, 838
cend, 838
clear, 837
const_local_iterator, 830
constructor, 830, 898
contains, 837
count, 837
depend, 833
dependable, 833
dependable_hint, 833
depend, 838
dependable, 838
equal_range, 837
erase, 836
erase_if, 898
extract, 836
find, 837
hash_function, 833
hasher, 830
insert, 834, 898
key_eq, 833
key_equal, 830
key_type, 830
load_factor, 838
local_iterator, 830
mapped_type, 830
max_bucket_count, 837
max_load_factor, 838
merge, 836
node_type, 830
rehash, 838
reserve, 839
value_type, 830
unordered_multiset, 887, 903
begin, 838
bucket, 838
bucket_count, 837
bucket_size, 838
cbegin, 838
cend, 838
clear, 837
const_local_iterator, 830
constructor, 830, 906, 907
contains, 837
count, 837
depend, 833
dependable, 833
dependable_hint, 833
depend, 838
equal_range, 837
## Index of library names

<table>
<thead>
<tr>
<th>Function/Class</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>erase</td>
<td>836</td>
</tr>
<tr>
<td>erase_if</td>
<td>907</td>
</tr>
<tr>
<td>extract</td>
<td>836</td>
</tr>
<tr>
<td>find</td>
<td>837</td>
</tr>
<tr>
<td>hash_function</td>
<td>833</td>
</tr>
<tr>
<td>hasher</td>
<td>830</td>
</tr>
<tr>
<td>insert</td>
<td>834</td>
</tr>
<tr>
<td>key_eq</td>
<td>833</td>
</tr>
<tr>
<td>key_equal</td>
<td>830</td>
</tr>
<tr>
<td>key_type</td>
<td>830</td>
</tr>
<tr>
<td>load_factor</td>
<td>838</td>
</tr>
<tr>
<td>local_iterator</td>
<td>830</td>
</tr>
<tr>
<td>mapped_type</td>
<td>830</td>
</tr>
<tr>
<td>max_bucket_count</td>
<td>837</td>
</tr>
<tr>
<td>max_load_factor</td>
<td>838</td>
</tr>
<tr>
<td>merge</td>
<td>836</td>
</tr>
<tr>
<td>node_type</td>
<td>830</td>
</tr>
<tr>
<td>rehash</td>
<td>838</td>
</tr>
<tr>
<td>reserve</td>
<td>839</td>
</tr>
<tr>
<td>value_type</td>
<td>830</td>
</tr>
<tr>
<td>unordered_set</td>
<td>887, 890</td>
</tr>
<tr>
<td>begin</td>
<td>838</td>
</tr>
<tr>
<td>bucket</td>
<td>838</td>
</tr>
<tr>
<td>bucket_count</td>
<td>837</td>
</tr>
<tr>
<td>bucket_size</td>
<td>838</td>
</tr>
<tr>
<td>cbegin</td>
<td>838</td>
</tr>
<tr>
<td>cend</td>
<td>838</td>
</tr>
<tr>
<td>clear</td>
<td>837</td>
</tr>
<tr>
<td>const_local_iterator</td>
<td>830</td>
</tr>
<tr>
<td>constructor</td>
<td>830, 902</td>
</tr>
<tr>
<td>contains</td>
<td>837</td>
</tr>
<tr>
<td>count</td>
<td>837</td>
</tr>
<tr>
<td>emplace</td>
<td>833</td>
</tr>
<tr>
<td>emplace_hint</td>
<td>833</td>
</tr>
<tr>
<td>end</td>
<td>838</td>
</tr>
<tr>
<td>equal_range</td>
<td>837</td>
</tr>
<tr>
<td>erase</td>
<td>836</td>
</tr>
<tr>
<td>erase_if</td>
<td>903</td>
</tr>
<tr>
<td>extract</td>
<td>836</td>
</tr>
<tr>
<td>find</td>
<td>837</td>
</tr>
<tr>
<td>hash_function</td>
<td>833</td>
</tr>
<tr>
<td>hasher</td>
<td>830</td>
</tr>
<tr>
<td>insert</td>
<td>834</td>
</tr>
<tr>
<td>key_eq</td>
<td>833</td>
</tr>
<tr>
<td>key_equal</td>
<td>830</td>
</tr>
<tr>
<td>key_type</td>
<td>830</td>
</tr>
<tr>
<td>load_factor</td>
<td>838</td>
</tr>
<tr>
<td>local_iterator</td>
<td>830</td>
</tr>
<tr>
<td>mapped_type</td>
<td>830</td>
</tr>
<tr>
<td>max_bucket_count</td>
<td>837</td>
</tr>
<tr>
<td>max_load_factor</td>
<td>838</td>
</tr>
<tr>
<td>merge</td>
<td>836</td>
</tr>
<tr>
<td>node_type</td>
<td>830</td>
</tr>
<tr>
<td>rehash</td>
<td>838</td>
</tr>
<tr>
<td>reserve</td>
<td>839</td>
</tr>
<tr>
<td>value_type</td>
<td>830</td>
</tr>
<tr>
<td>unreachable_sentinel</td>
<td>928</td>
</tr>
<tr>
<td>unreachable_sentinel_t</td>
<td>973</td>
</tr>
<tr>
<td>operator==</td>
<td>973</td>
</tr>
</tbody>
</table>

### V

<table>
<thead>
<tr>
<th>Function/Class</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>va_arg</td>
<td>550</td>
</tr>
<tr>
<td>va_copy</td>
<td>550</td>
</tr>
<tr>
<td>va_end</td>
<td>500, 550</td>
</tr>
<tr>
<td>va_list</td>
<td>500, 550</td>
</tr>
<tr>
<td>va_start</td>
<td>550</td>
</tr>
<tr>
<td>valarray</td>
<td>1213, 1225</td>
</tr>
<tr>
<td>abs</td>
<td>1221</td>
</tr>
<tr>
<td>acos</td>
<td>1221</td>
</tr>
<tr>
<td>apply</td>
<td>1219</td>
</tr>
<tr>
<td>asin</td>
<td>1221</td>
</tr>
<tr>
<td>atan</td>
<td>1221</td>
</tr>
<tr>
<td>atan2</td>
<td>1221</td>
</tr>
<tr>
<td>begin</td>
<td>1228</td>
</tr>
</tbody>
</table>
constructor, 1214, 1215
cos, 1221

cosh, 1221
cshift, 1219
destructor, 1215
deg, 1228

dexp, 1221

dlog, 1221
dlog10, 1221

dmax, 1219
dmin, 1219

doperator!, 1217
doperator*, 1219, 1220
doperator**=, 1218
doperator+, 1217, 1219, 1220
doperator+=, 1218
doperator-, 1217, 1219, 1220
doperator-=, 1218
doperator/, 1219, 1220
doperator/=, 1218
doperator<, 1220, 1221
doperator<<, 1219, 1220
doperator<<<, 1218
doperator<, 1220, 1221
doperator=, 1215
doperator**, 1220, 1221
doperator>>, 1219, 1220
doperator>>, 1219, 1220
doperator[](), 1216, 1217
doperator[]%, 1219, 1220
doperator%!, 1218
doperator&!, 1219, 1220
doperator&&, 1218
doperator&&, 1220, 1221
doperator~!, 1219, 1220
doperator!, 1217
doperator<, 1219, 1220

dpow, 1221

dresize, 1219
dshift, 1219
dsin, 1221
dsinh, 1221
dsize, 1218
dsqrt, 1221
dsum, 1219
dswap, 1218, 1222
dtan, 1221
tanh, 1221

dvalid
dfuture, 1623
dpackaged_task, 1628
dshared_future, 1625
dvalue
derror_code, 576
derror_condition, 577
dleap_second, 1325
doptional, 608, 609
dregex_traits, 1516
dvalue_comp
dordered associative containers, 821
dvalue_compare
dordered associative containers, 820
dvalue_or
doptional, 609
dvalue_type
dallocator, 648
datomic, 1552
datomic_ref, 1546
dbasic_string, 768
dbasic_string_view, 792
dcomplex, 1164
gslice_array, 1225
dindirect_array, 1227
dinteger_sequence, 585
dintegral_constant, 715
dmask_array, 1226
doptional, 601
dordered associative containers, 820
dpath, 1465
dpolymorphic_allocator, 674
dscoped_allocator_adaptor, 681
dslice_array, 1223

dunordered associative containers, 830
dvalueless_by_exception
dvariant, 619

dvariant, 614
dconstructor, 615, 616
destructor, 617
demplace, 618, 619
dget, 620
dget_if, 620, 621
dhash, 623
dholds_alternative, 620
dindex, 619
doperator!, 621
doperator<, 621
doperator<, 621
doperator<, 621
doperator<, 621
doperator<, 621
doperator<, 621
doperator<, 621
doperator<, 621
doperator<, 621
doperator<, 621

dswap, 619, 622
dvalueless_by_exception, 619

dvisit, 622

dvariant_alternative, 620

dvariant_alternative_t, 612

dvariant_size, 620

dvariant_size_v, 612

dvector, 861
dcapacity, 864
dconstructor, 863

Index of library names 1838
data, 865
erase, 865
erase_if, 866
insert, 865
operator<, 863
operator==, 863
reserve, 864
resize, 864, 865
shrink_to_fit, 864
swap, 864
vector<bool>, 866

flip, 868
swap, 868
vformat, 749
vformat_to, 750
vfscanf, 1503
vfprintf, 802
vwscanf, 802
view, 991

basic_istringstream, 1435
basic_ostringstream, 1438
basic_stringbuf, 1431
basic_stringstream, 1441
view_interface, 993
back, 994
front, 994
viewable_range, 993
visit, 622
variant, 622
visit_format_arg, 757
void_pointer
 allocator_traits, 647
 scoped_allocator_adaptor, 681
void_t, 712
vprintf, 1503
vscanf, 1503
vsnprintf, 1503
vsscanf, 1503
vwprintf, 1503
vwscanf, 1503

W

atomic, 1556
atomic<floating-point>, 1556
atomic<integral>, 1556
atomic<shared_ptr<T>>, 1565
atomic<T>, 1556
atomic<weak_ptr<T>>, 1567
atomic_flag, 1569
atomic_ref<T>, 1548
barrier, 1615
condition_variable, 1605, 1606
condition_variable_any, 1609

future, 1623
latch, 1614
shared_future, 1625
wait_for
 condition_variable, 1606, 1607
 condition_variable_any, 1609, 1610
 future, 1623
 shared_future, 1625
wait_until
 condition_variable, 1606, 1607
 condition_variable_any, 1609, 1610
 future, 1623
 shared_future, 1625
wbuffer_convert, 1700
 constructor, 1701
 destructor, 1701
 rdbuf, 1701
 state, 1701
 state_type, 1701
wcerr, 1379
WCHAR_MAX, 802
WCHAR_MIN, 802
wcin, 1379
wclog, 1379
wcout, 1379
wcrstombs, 804
wctomb, 802, 804
wsccat, 802
wcscmp, 802
wcscoll, 802
wcscpy, 802
wcscspn, 802
wcsftime, 802
wcslen, 802
wcsncpy, 802
wcspbrk, 802
wcsrchr, 802
wcsrtombs, 802
wcsspn, 802
wcsstr, 802
wcstod, 802
wcstof, 802
wcstoiimax, 1504
wcstok, 802
wcslen, 802
wcstold, 802
wcstoll, 802
wcstombs, 507, 804
wcstoul, 802
wcstoull, 802
wcstoumax, 1504
wcsxfrm, 802
wctob, 802
wctomb, 507, 804
wctrans, 801
wctrans_t, 801
<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wctype, 801</td>
<td></td>
</tr>
<tr>
<td>wctype_t, 801</td>
<td></td>
</tr>
<tr>
<td>weak_from_this</td>
<td></td>
</tr>
<tr>
<td>enable_shared_from_this, 672</td>
<td></td>
</tr>
<tr>
<td>weak_order, 544</td>
<td></td>
</tr>
<tr>
<td>weak_ordering, 540</td>
<td></td>
</tr>
<tr>
<td>equivalent, 540</td>
<td></td>
</tr>
<tr>
<td>greater, 540</td>
<td></td>
</tr>
<tr>
<td>less, 540</td>
<td></td>
</tr>
<tr>
<td>operator partial_ordering, 541</td>
<td></td>
</tr>
<tr>
<td>operator&lt;, 541</td>
<td></td>
</tr>
<tr>
<td>operator&lt;=, 541</td>
<td></td>
</tr>
<tr>
<td>operator==, 541</td>
<td></td>
</tr>
<tr>
<td>operator&gt;, 541</td>
<td></td>
</tr>
<tr>
<td>operator&gt;=, 541</td>
<td></td>
</tr>
<tr>
<td>weak_ptr, 661, 668, 672</td>
<td></td>
</tr>
<tr>
<td>constructor, 669</td>
<td></td>
</tr>
<tr>
<td>destructor, 670</td>
<td></td>
</tr>
<tr>
<td>expired, 670</td>
<td></td>
</tr>
<tr>
<td>lock, 670</td>
<td></td>
</tr>
<tr>
<td>operator=, 670</td>
<td></td>
</tr>
<tr>
<td>owner_before, 670</td>
<td></td>
</tr>
<tr>
<td>reset, 670</td>
<td></td>
</tr>
<tr>
<td>swap, 670</td>
<td></td>
</tr>
<tr>
<td>use_count, 670</td>
<td></td>
</tr>
<tr>
<td>weakly_canonical, 1502</td>
<td></td>
</tr>
<tr>
<td>weakly_incrementable, 937</td>
<td></td>
</tr>
<tr>
<td>weekday, 1288</td>
<td></td>
</tr>
<tr>
<td>c_encoding, 1289</td>
<td></td>
</tr>
<tr>
<td>constructor, 1288, 1289</td>
<td></td>
</tr>
<tr>
<td>from_stream, 1290</td>
<td></td>
</tr>
<tr>
<td>iso_encoding, 1289</td>
<td></td>
</tr>
<tr>
<td>ok, 1289</td>
<td></td>
</tr>
<tr>
<td>operator+, 1289, 1290</td>
<td></td>
</tr>
<tr>
<td>operator++, 1289</td>
<td></td>
</tr>
<tr>
<td>operator=, 1289</td>
<td></td>
</tr>
<tr>
<td>operator-, 1290</td>
<td></td>
</tr>
<tr>
<td>operator--, 1289</td>
<td></td>
</tr>
<tr>
<td>operator==, 1289</td>
<td></td>
</tr>
<tr>
<td>operator&lt;=, 1290</td>
<td></td>
</tr>
<tr>
<td>operator&gt;&gt;=, 1289</td>
<td></td>
</tr>
<tr>
<td>operator[] , 1289</td>
<td></td>
</tr>
<tr>
<td>weekday_indexed, 1291</td>
<td></td>
</tr>
<tr>
<td>weekday_last, 1292</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday, 1304</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday_last, 1306</td>
<td></td>
</tr>
<tr>
<td>weekday_indexed, 1290</td>
<td></td>
</tr>
<tr>
<td>constructor, 1291</td>
<td></td>
</tr>
<tr>
<td>index, 1291</td>
<td></td>
</tr>
<tr>
<td>month_weekday, 1294</td>
<td></td>
</tr>
<tr>
<td>ok, 1291</td>
<td></td>
</tr>
<tr>
<td>operator&lt;&lt;, 1291</td>
<td></td>
</tr>
<tr>
<td>operator==, 1291</td>
<td></td>
</tr>
<tr>
<td>weekday, 1291</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday, 1304</td>
<td></td>
</tr>
<tr>
<td>weekday_last, 1291</td>
<td></td>
</tr>
<tr>
<td>constructor, 1292</td>
<td></td>
</tr>
<tr>
<td>month_weekday_last, 1295</td>
<td></td>
</tr>
</tbody>
</table>

Index of library names
Index of library names
year, 1304
year_month_weekday_last, 1305
    constructor, 1306
month, 1306
ok, 1307
operator local_days, 1307
operator sys_days, 1307
operator+, 1307
operator+=, 1306
operator-, 1307
operator-=, 1306
operator<<, 1307
operator==, 1307
weekday, 1306
weekday_last, 1307
year, 1306
years, 1245
yield
    this_thread, 1585

Z
zero
    duration, 1263
    duration_values, 1260
zoned_time, 1320
    constructor, 1322–1323
    get_info, 1324
    get_local_time, 1323
    get_sys_time, 1324
    get_time_zone, 1323
    operator local_time, 1323
    operator sys_time, 1323
    operator<<, 1324
    operator=, 1323
    operator==, 1324
zoned_traits, 1320
zoned_traits<const time_zone*>*
    default_zone, 1320
    locate_zone, 1320

Index of library names 1842
Index of library concepts

The bold page number for each entry is the page where the concept is defined. Other page numbers refer to pages where the concept is mentioned in the general text.

**advanceable**, 1000, **1001**, 1001–1005
assignable_from, 555, **558**, 558, 559, 564, 948, 951, 964–966, 969, 970, 1009

**bidirectional_iterator**, 923, 926, 929, **940**, 940, 941, 951–953, 961, 969, 971, 987, 988, 992, 996, 997, 1058, 1059, 1065, 1066, 1073, 1075, 1084, 1100, 1101, 1111, 1123, 1124, 1126, 1140, 1141

**bidirectional_range**, 985, **992**, 992, 994, 1011, 1012, 1015–1017, 1027–1029, 1037, 1040, 1042, 1058, 1059, 1065, 1066, 1073, 1075, 1084, 1100, 1101, 1111, 1124, 1127, 1140, 1141

**boolean-testable**, 482, 543, 561, **562**, 562, 564
**boolean-testable-impl**, **561**, 561, 562

**borrowed_range**, 918, 983, **990**, 990, 991, 995, 997, 998

can-reference, **922**, 922, 923, 932, 938, 984, 1013, 1015, 1018

common_range, **992**, 992, 1036

common_reference_with, 544, 554, **557**, 557, 559

common_with, 555, **557**, 558, 969, 972

**compares-as**, **543**, 543, 544

**constructible_from**, 555, **560**, 560, 636, 637, 932, 947, 948, 967, 995, 999, 1000, 1031, 1036, 1037, 1578

**contiguous_iterator**, 795, 799, 810, 917–919, 921, 924, 929, **941**, 941, 989, 1035, 1228

contiguous_range, **918**, 919, **992**, 992, 1009

**convertible-to-non-slicing**, **994**, 995–997

**convertible-to**, 114, 401, 554, **557**, 561, 932, 933, 964–966, 969, 970, 995, 997, 1001, 1010, 1015, 1016, 1018, 1019, 1021–1023, 1027, 1028, 1030, 1032, 1033, 1035, 1040, 1041, 1043

**copy_constructible**, 555, **561**, 561, 564, 946, 947, 961, 983, 984, 999, 1008, 1013, 1015, 1018, 1026, 1036, 1038, 1060, 1063, 1088, 1089, 1103, 1107

**copyable**, **564**, 932, 948, 965, 1011, 1012, 1027, 1036, 1080, 1081, 1135–1137

cpp17-bidirectional_iterator, **933**, 933
cpp17-forward_iterator, **932**, 933
cpp17-input_iterator, **932**, 932, 933
cpp17-iterator, **932**, 932, 933
cpp17-random-access_iterator, **933**, 933

decrecrementable, 1000, **1001**, 1001, 1003, 1004
default_initializable, 555, **561**, 991, 1008
dereferenceable, **922**, 922, 923, 936, 965, 966, 969, 970
derived_from, 554, **556**, 556, 939–941, 953, 961, 966, 994, 995, 1012, 1016, 1028, 1034
destructible, 480, 555, **560**, 560, 638, 1578
equality_comparable, 555, **563**, 563, 932, 936, 938, 939, 1015, 1027, 1041
equality_comparable_with, **563**, 563

equivalence_relation, 556, **565**, 565, 946, 947

floating_point, **558**, 1244
formal_iterator, 923, 929, **940**, 940, 949, 962, 966, 967, 969, 971, 989–992, 1052, 1053, 1056, 1057, 1065, 1066, 1070–1072, 1074, 1082, 1091, 1092, 1096–1098, 1110, 1112, 1114, 1119, 1121, 1122, 1125, 1137, 1138, 1155


has-arrow, **993**, 1011, 1012, 1027, 1029

has-tuple-element, **1039**, 1039, 1040

incrementable, 923, **938**, 938, 940, 1001, 1003, 1004

indirect_binary_predicate, **946**, 948, 1051, 1053, 1061–1064, 1090, 1092, 1093, 1104, 1105, 1107, 1108

indirect_equivalence_relation, **946**, 1056, 1064, 1065, 1096, 1109, 1110

indirect_strict_weak_order, **947**, 949, 1069–1072, 1075, 1080–1083, 1118, 1119, 1121, 1122, 1127, 1128, 1134–1139

indirect_unary_predicate, **946**, 964, 984, 1010, 1011, 1013, 1021, 1022, 1024, 1049–1051, 1053, 1058, 1061–1064, 1072–1074, 1087, 1088, 1090, 1093, 1100, 1104–1109, 1123–1125

indirectly-readable-impl, **936**

indirectly_comparable, 924, 947, **948**, 948, 984, 1031–1034, 1052, 1054–1057, 1091, 1092, 1094, 1095, 1097, 1098

indirectly_copyable, 924, 947, **948**, 948, 949, 1057, 1058, 1061, 1062, 1064–1067, 1069, 1073, 1081, 1099, 1100, 1105, 1106, 1108–1113, 1118, 1124, 1135

© ISO/IEC

1843
© ISO/IEC
indirectly_copyable_storable, 948, 948,
1065, 1110, 1136, 1137
indirectly_movable, 924, 947, 947, 964, 1059,
1101
indirectly_movable_storable, 935, 947,
947–949
indirectly_readable, 923, 924, 931, 936, 936,
939, 946–948, 967
indirectly_regular_unary_invocable, 924,
946, 947
indirectly_swappable, 924, 947, 948, 948, 949,
953, 956, 961, 963, 965, 968, 970, 972,
1011, 1013, 1016, 1018, 1034, 1035, 1059,
1102
indirectly_unary_invocable, 946, 1050, 1051,
1088, 1089
indirectly_writable, 923, 929, 936, 936, 939,
947, 948, 1060, 1061, 1063, 1103–1105,
1107
input_iterator, 636, 637, 923, 927–929, 939,
939, 940, 949, 961, 964–966, 968, 970,
972, 992, 993, 1049–1055, 1057–1062,
1064, 1065, 1069, 1072–1078, 1083, 1087,
1088, 1090, 1092–1095, 1099–1105,
1108–1110, 1118, 1123, 1124, 1126–1131,
1139, 1154
input_or_output_iterator, 923, 925, 927, 929,
938, 938, 939, 951, 952, 965, 969, 982,
986–988, 990, 995, 996, 1063, 1107
input_range, 636, 637, 984, 985, 992, 992, 1006,
1010, 1011, 1013, 1015, 1018, 1021, 1022,
1024–1027, 1030–1034, 1049–1055,
1057–1062, 1064, 1065, 1067, 1069,
1072–1078, 1081, 1083, 1087, 1088, 1090,
1092–1095, 1099–1106, 1108–1110, 1113,
1118, 1123, 1124, 1126, 1128–1131,
1135–1137, 1139, 1157
integral, 339, 340, 558, 937
invocable, 484, 556, 564, 564, 924, 946, 1063,
1107, 1177, 1578
iterator-sentinel-pair , 995, 996
mergeable, 925, 947, 949, 949, 1074, 1076–1078,
1126, 1128–1131
movable, 564, 564, 937, 947, 991
move_constructible, 555, 559, 561, 561
no-throw-forward-iterator , 635–637, 1155
no-throw-forward-range , 635–637, 1155
no-throw-input-iterator , 635, 638, 1154
no-throw-input-range , 635, 638, 1155
no-throw-sentinel-for , 635–638, 1154
not-same-as , 993, 994, 995, 997, 1009, 1010
output_iterator, 750, 754, 923, 929, 939, 939,
940, 992, 1061, 1062, 1105, 1106
output_range, 992, 992, 1062, 1063, 1106, 1107

N4868

pair-like-convertible-from , 995, 995, 997
partially-ordered-with , 543, 543, 544, 563
permutable, 925, 947, 948, 949, 949, 1063–1067,
1072, 1073, 1107–1109, 1111, 1112, 1114,
1123, 1124
predicate, 556, 564, 946
random_access_iterator, 924, 929, 939, 940,
941, 941, 951, 953, 961, 969–972, 990,
992, 1035, 1067–1069, 1071, 1078–1080,
1114, 1116–1118, 1120, 1132–1135
random_access_range, 992, 992, 994, 1015–1020,
1023, 1024, 1036, 1037, 1040–1043, 1069,
1079, 1080, 1114, 1116–1118, 1120,
1132–1135
range, 925, 952, 981–983, 990, 990–992, 995,
1008, 1009, 1011, 1020, 1038, 1046
regular, 564, 564, 937
regular_invocable, 556, 564, 564, 946
relation, 556, 564, 565
same-as-impl , 556, 556
same_as, 116, 543, 554, 556, 556–558, 927, 932,
933, 936–941, 951, 952, 965, 982, 992,
993, 998, 1000–1002, 1065, 1110, 1177
semiregular, 484, 564, 564, 927, 938, 964, 983,
1000, 1002, 1005
sentinel_for, 636, 637, 923, 925, 927, 938,
938–940, 951, 952, 960, 962–965, 967,
982, 986–988, 990, 995, 996, 1049–1080,
1082–1084, 1087, 1088, 1090–1112, 1114,
1116–1135, 1137–1141, 1154
signed_integral, 558, 558, 937
simple-view , 993, 1020, 1022–1024, 1026, 1039
sized_range, 918, 952, 982, 990, 991, 991,
994–997, 1009, 1014, 1019, 1020, 1023,
1024, 1036–1038
sized_sentinel_for, 795, 799, 918, 923, 926,
939, 939, 941, 951, 952, 960, 963, 965,
968, 982, 989, 994–997, 1002, 1006, 1018,
1019, 1096, 1097
sortable, 925, 947, 949, 949, 1068, 1069, 1071,
1075, 1078, 1079, 1084, 1116–1118, 1120,
1126, 1127, 1132–1134, 1140, 1141
stream-extractable , 1006, 1006, 1007
strict_weak_order, 556, 565, 565, 947
swappable, 555, 559, 559
swappable_with, 559, 559, 560, 935
three_way_comparable, 538, 543, 543, 544, 908,
909, 911, 915, 1003, 1005, 1016, 1018,
1041, 1042, 1265
three_way_comparable_with, 544, 544, 600,
610, 611, 639, 657, 658, 926, 927, 956,
963, 1247, 1256, 1270, 1326
tiny-range , 984, 1030, 1031–1035
totally_ordered, 480, 543, 555, 563, 563, 933,
937, 941, 1001, 1003, 1005

pair-like , 995, 995
Index of library concepts

1844


totally_ordered_with, 544, 563, 563, 695, 696, 1002

uniform_random_bit_generator, 1067, 1114, 1177, 1177

unsigned_integral, 558, 558, 937, 1177


viewable_range, 983, 992, 993, 1008, 1036

weakly-equality-comparable-with, 543, 544, 562, 563, 938, 983, 1000, 1002, 1005


Index of library concepts 1845
Index of implementation-defined behavior

The entries in this index are rough descriptions; exact specifications are at the indicated page in the general text.

#pragma, 474
additional execution policies supported by parallel algorithms, 738, 1048
additional file_type enumerators for file systems supporting additional types of file, 1479
additional formats for time_get::do_get_date, 1364
additional supported forms of preprocessing directive, 462
algorithms for producing the standard random number distributions, 1193
alignment, 68
alignment additional values, 69
alignment of bit-fields within a class object, 283
allocation of bit-fields within a class object, 283
any use of an invalid pointer other than to perform indirection or deallocate, 65
argument values to construct ios_base::failure, 1392
assignability of placeholder objects, 701
behavior of iostream classes when traits::pos_type is not streampos or when traits::off_type is not streamoff, 1376
behavior of non-standard attributes, 241
behavior of strstreambuf::setbuf, 1689
bits in a byte, 58
choice of larger or smaller value of floating-point-literal, 24
concatenation of some types of string-literals, 25
conversions between pointers and integers, 125
converting characters from source character set to execution character set, 13
converting function pointer to object pointer and vice versa, 125
default configuration of a pool, 679
default next_buffer_size for a monotonic_buffer_resource, 680
default number of buckets in unordered_map, 892
default number of buckets in unordered_multimap, 898
default number of buckets in unordered_multiset, 907
default number of buckets in unordered_set, 902, 903
defining main in freestanding environment, 86
definition and meaning of __STDC_VERSION__, 476
definition of NULL, 508, 1679
derived type for typeid, 122
diagnostic message, 4
dynamic initialization of static inline variables before main, 89
dynamic initialization of static variables before main, 88, 89
dynamic initialization of thread-local variables before entry, 89
effect of calling associated Laguerre polynomials with \( n \geq 128 \) or \( m \geq 128 \), 1239
effect of calling associated Legendre polynomials with \( l \geq 128 \), 1239
effect of calling basic_filebuf::setbuf with nonzero arguments, 1446
effect of calling basic_filebuf::sync when a get area exists, 1447
effect of calling basic_streambuf::setbuf with nonzero arguments, 1433
effect of calling cylindrical Bessel functions of the first kind with \( \nu \geq 128 \), 1240
effect of calling cylindrical Neumann functions with \( \nu \geq 128 \), 1241
effect of calling Hermite polynomials with \( n \geq 128 \), 1242
effect of calling io_base::sync_with_stdio after any input or output operation on standard streams, 1386
effect of calling irregular modified cylindrical Bessel functions with \( n \geq 128 \), 1240
effect of calling Laguerre polynomials with \( n \geq 128 \), 1242
effect of calling Legendre polynomials with \( l \geq 128 \), 1242
effect of calling regular modified cylindrical Bessel functions with \( n \geq 128 \), 1240
effect of calling spherical associated Legendre functions with \( l \geq 128 \), 1243
effect of calling spherical Bessel functions with \( n \geq 128 \), 1243
effect of calling spherical Neumann functions with \( n \geq 128 \), 1243
effect of filesystem::copy, 1491
effect on C locale of calling locale::global, 1341
encoding of universal character name not in execution character set, 23
error_category for errors originating outside the operating system, 505
exception type when `random_device` constructor fails, 1191
exception type when `random_device::operator()` fails, 1191
exception type when `shared_ptr` constructor fails, 661
exceptions thrown by standard library functions that have a potentially-throwing exception specification, 505
execution character set and execution wide-character set, 15
exit status, 522
extended signed integer types, 74
file type of the file argument of `filesystem::status`, 1502
formatted character sequence generated by `time_put::do_put` in C locale, 1367
forward progress guarantees for implicit threads of parallel algorithms (if not defined for `thread`), 1047
growth factor for `monotonic_buffer_resource`, 680, 681
headers for freestanding implementation, 487
how `random_device::operator()` generates values, 1191
how the set of importable headers is determined, 252
interactive device, 11
interpretation of the path character sequence with format `path::auto_format`, 1479
largest supported value to configure the largest allocation satisfied directly by a pool, 678
largest supported value to configure the maximum number of blocks to replenish a pool, 678
linkage of `main`, 87
linkage of names from C standard library, 488
linkage of objects between C++ and other languages, 240
locale names, 1340
lvalue-to-rvalue conversion of an invalid pointer value, 96
manner of search for included source file, 466
mapping from name to catalog when calling `messages::do_open`, 1373
mapping from physical source file characters to basic source character set, 14, 1600
mapping header name to header or external source file, 17
mapping of pointer to integer, 125
mapping physical source file characters to basic source character set, 13
mapping to message when calling `messages::do_get`, 1373
maximum depth of recursive template instantiations, 423
maximum size of an allocated object, 132, 528
meaning of ‘’, ‘\’, ‘/’, ‘*’, or ‘/’ in a `q-char-sequence` or an `h-char-sequence`, 17
meaning of `asm` declaration, 237
meaning of attribute declaration, 165
meaning of dot-dot in `root-directory`, 1466
negative value of `character-literal` in preprocessor, 465
nesting limit for `#include` directives, 466
NTCTS in `basic_ostream<charT, traits>& operator<<(nullptr_t)`, 1420
number of placeholders for bind expressions, 686, 701
number of threads in a program under a freestanding implementation, 82
numeric values of `character-literals` in `#if` directives, 464
operating system on which implementation depends, 1458
parameters to `main`, 87
passing argument of class type through ellipsis, 119
physical source file characters, 13
presence and meaning of `native_handle_type` and `native_handle`, 1571
range defined for `character-literals`, 23
rank of extended signed integer type, 79
required alignment for `atomic_ref` type’s operations, 1546, 1549–1551
required libraries for freestanding implementation, 10
resource limits on a message catalog, 1373
result of `filesystem::file_size`, 1496
result of inexact floating-point conversion, 98
return value of `bad_alloc::what`, 528
return value of `bad_any_cast::what`, 624
return value of `bad_array_new_length::what`, 528
return value of `bad_cast::what`, 531
return value of `bad_exception::what`, 534
return value of `bad_function_call::what`, 701
return value of `bad_optional_access::what`, 609
return value of `bad_typeid::what`, 531
return value of `bad_variant_access::what`, 623
return value of `bad_weak_ptr::what`, 658
return value of `char_traits<char16_t>::eof`, 764
return value of `char_traits<char32_t>::eof`, 764
return value of `char_traits< wchar_t >::eof`, 764
return value of `exception::what`, 534
return value of `type_info::name()`, 530
search locations for "" header, 466
search locations for <> header, 465
semantics of an access through a volatile glvalue, 175
semantics of linkage specification on templates, 362
semantics of linkage specifiers, 237
semantics of non-standard escape sequences, 22
semantics of parallel algorithms invoked with implementation-defined execution policies, 1048
semantics of token parameter and default token value used by random_device constructors, 1191
sequence of places searched for a header, 465
set of character types that iostreams templates can be instantiated for, 1339, 1376
signedness of char, 176
sizeof applied to fundamental types other than char, signed char, and unsigned char, 130
stack unwinding before call to std::terminate, 456, 460
startup and termination in freestanding environment, 86
strict total order over pointer values, 5
string resulting from __func__, 215
support for always lock-free integral atomic types in freestanding environments, 487
support for extended alignments, 69
support for module-import-declarations with non-C++ language linkage, 238
supported multibyte character encoding rules, 763
supported root-names in addition to any operating system dependent root-names, 1466
text of __DATE__ when date of translation is not available, 474
text of __TIME__ when time of translation is not available, 474
threads and program points at which deferred dynamic initialization is performed, 88, 89
type aliases atomic_signed_lock_free and atomic_unsigned_lock_free in freestanding environments, 487
type of a directory-like file, 1486, 1488
type of array::const_iterator, 843
type of array::iterator, 843
type of basic_string::const_iterator, 769
type of basic_string::iterator, 769
type of basic_string_view::const_iterator, 792, 795
type of default_random_engine, 1190
type of deque::const_iterator, 846
type of deque::iterator, 846
type of forward_list::const_iterator, 850
type of forward_list::iterator, 850
type of list::const_iterator, 856
type of list::iterator, 856

Index of impl.-def. behavior

© ISO/IEC

N4868

type of map::const_iterator, 870
type of map::iterator, 870
type of multimap::const_iterator, 876
type of multimap::iterator, 876
type of multiset::const_iterator, 883
type of multiset::iterator, 883
type of ptrdiff_t, 140, 509
type of regex_constants::error_type, 1514
type of regex_constants::match_flag_type, 1513
type of set::const_iterator, 879
type of set::iterator, 879
type of size_t, 509
type of span::iterator, 916, 921
type of syntax_option_type, 1512
type of unordered_map::const_iterator, 888
type of unordered_map::const_local_iterator, 888
type of unordered_map::iterator, 888
type of unordered_map::local_iterator, 888
type of unordered_multimap::const_iterator, 894
type of unordered_multimap::const_local_iterator, 895
type of unordered_multimap::iterator, 895
type of unordered_multiset::const_iterator, 903
type of unordered_multiset::const_local_iterator, 904
type of unordered_multiset::iterator, 903
type of unordered_multiset::local_iterator, 903
type of unordered_set::const_iterator, 899
type of unordered_set::const_local_iterator, 899
type of unordered_set::iterator, 899
type of unordered_set::local_iterator, 899
type of vector::const_iterator, 861
type of vector::iterator, 861
type of vector<bool>::const_iterator, 866
type of vector<bool>::iterator, 866

underlying type for enumeration, 222
underlying type of bool, 76
underlying type of char, 75
underlying type of wchar_t, 75

unit suffix when Period::type is micro, 1267

value for least_max_value default template argument of counting_semaphore, 1611
value of bit-field that cannot represent assigned value, 147
incremented value, 121
value of character-literal outside range of corresponding type, 23
value of ctype<char>::table_size, 1346
value of has-attribute-expression for non-standard attributes, 464
value of multicharacter literal, 22
value of pow(0,0), 1169
value of result of inexact integer to floating-point conversion, 98
value of wide-character literal containing multiple characters, 22
value of wide-character literal with single c-char that is not in execution wide-character set, 22
value representation of floating-point types, 76
value representation of pointer types, 77
values of a trivially copyable type, 73
values of various ATOMIC_..._LOCK_FREE macros, 1545
whether <cfenv> functions can be used to manage floating-point status, 1162
whether a given atomic type's operations are always lock free, 1553, 1554, 1557, 1559, 1560, 1563, 1565
whether a given atomic_ref type's operations are always lock free, 1546, 1549–1551
whether an implementation has relaxed or strict pointer safety, 68
whether functions from Annex K of the C standard library are declared when C++ headers are included, 487
whether get_pointer_safety returns pointer_safety::relaxed or pointer_safety::preferred if the implementation has relaxed pointer safety, 643
whether locale object is global or per-thread, 1338
whether pragma FENV_ACCESS is supported, 1162
whether rand may introduce a data race, 1210
whether sequence pointers are copied by basic_filebuf move constructor, 1443
whether sequence pointers are copied by basic_stringbuf move constructor, 1430
whether sequence pointers are initialized to null pointers, 1429
whether source file inclusion of importable header is replaced with import directive, 466
whether source of translation units must be available to locate template definitions, 14
whether stack is unwound before calling the function std::terminate when a noexcept specification is violated, 460
whether the implementation is hosted or freestanding, 487
whether the lifetime of a parameter ends when the callee returns or at the end of the enclosing full-expression, 118
whether the sources for module units and header units on which the current translation unit has an interface dependency are required to be available during translation, 14
whether the thread that executes main and the threads created by std::thread or std::jthread provide concurrent forward progress guarantees, 86
whether time_get::do_get_year accepts two-digit year numbers, 1365
whether values are rounded or truncated to the required precision when converting between time_t values and time_point objects, 1271
which functions in the C++ standard library may be recursively reentered, 503
which non-standard-layout objects containing no data are considered empty, 60
which scalar types have unique object representations, 723
width of integral type, 75